UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Verificación formal de programas para el procesamiento de imágenes digitales

Autor/es

BORJA JIMENO SOTO

Director/es

ANA ROMERO IBÁÑEZ y JOSE DIVASÓN MALLAGARAY ,

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2017-18

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Verificación formal de programas para el procesamiento de imágenes digitales

Realizado por:

Borja Jimeno Soto

Tutelado por:

Jose Divasón Mallagaray

Ana Romero Ibáñez

Logroño, julio, 2018

Resumen

En este trabajo se muestran, en primer lugar, los resultados de la investigación realizada sobre las herramientas actualmente utilizadas para la verificación formal de programas escritos en Java. Se han comparado las herramientas *Krakatoa*, *KeY y OpenJML.* Se han comparado sus características y elegido una para desarrollar la segunda parte del trabajo.

En segundo lugar, se ha realizado un estudio teórico de algunos algoritmos de umbralización utilizados hoy en día.

Para finalizar, se ha realizado la especificación y verificación de algunos algoritmos de umbralización implementados en Java y se han recogido las dificultades de especificar código ya escrito.

Abstract

This work presents the results of a research about tools devoted to formally verify Java programs. Firstly, I show a review of three verification tools for Java programs: Krakatoa, KeY and OpenJML. Their features are compared and one of them is chosen to carry out the second part of the work.

Secondly, I have studied some of the current algorithms used in thresholding process.

Finally, some of such Java algorithms have been specified and formally verified, showing the difficulties which arose when specifying legacy code.

1. Introducción

Esta sección explicará los antecedentes que han llevado a la realización de este estudio, los objetivos propuestos para este trabajo, las personas que han intervenido en su desarrollo y la descripción de algunos conceptos previos necesarios.

1.1 Antecedentes

Hoy en día el uso de procesamiento de imágenes digitales está presente en casi todos los ámbitos de la sociedad. Podemos verlo en la monitorización del tráfico, reconocimiento automático de caracteres (OCR) o incluso como una herramienta que facilita y acelera numerosas investigaciones científicas.

En muchos de estos casos se trabaja con algoritmos que utilizan imágenes en blanco y negro. Estas imágenes se obtienen transformando imágenes en escala de grises. En el tratamiento de imágenes digitales, a este proceso se le conoce como binarización. Este proceso requiere el uso de un umbral adecuado para cada imagen. Calcular este umbral es muy complejo ya que depende de cada imagen y por ello existen muchos algoritmos distintos para su cálculo.

Si las implementaciones de los algoritmos de umbralización no son correctas las conclusiones que se obtengan en las investigaciones que hacen uso de ellos pueden ser erróneas. En este trabajo se tratará de verificar formalmente algunos de estos algoritmos.

Una de las herramientas utilizadas en el mundo de la investigación y del tratamiento digital de imágenes es *ImageJ[1]*. Esta es una herramienta *opensource* escrita en Java. Además, es extensible mediante el uso de *plugins* y scripts. También cuenta con una gran cantidad de usuarios y desarrolladores que hacen grandes aportes a la herramienta.

El problema aparece cuando un investigador quiere hacer uso de una de estas aportaciones. Debe confiar en que el algoritmo que se ha implementado realmente hace lo que se espera de él. El desarrollador ha podido realizar pruebas como test unitarios y de integración, pero esto no es suficiente para garantizar la corrección de los algoritmos.

Una solución para esto es la verificación formal de los programas que nos permite especificar nuestros algoritmos y comprobar su correcto funcionamiento bajo unas condiciones concretas.

Para la verificación formal existen numerosas herramientas que nos ayudan en este proceso.

1.2 Objetivos

El objetivo de este TFG es el de realizar un estudio de diferentes herramientas disponibles para la verificación formal de programas escritos en Java, la realización de un estudio de los distintos algoritmos de umbralización existentes y, por último, se procederá a la verificación formal de algunos de estos algoritmos.

1.3 Integrantes

Dado que este trabajo no se está desarrollando dentro de una empresa, los únicos participantes en el mismo aparte del estudiante son los dos tutores: Jose Divasón, Ana Romero. Ambos son profesores del Departamento de Matemáticas y Computación de la Universidad de La Rioja.

1.4 Conceptos previos

Para la comprensión del trabajo es necesario tener claros algunos conceptos sobre el procesamiento digital de imágenes y la verificación formal de algoritmos.

1.4.1 Procesamiento digital de imágenes

En el análisis de imágenes digitales aparece el proceso de *segmentación* [2]. Este proceso consiste en la partición de una imagen en regiones homogéneas. La segmentación se realiza después de haber realizado una corrección de la imagen.

La segmentación se puede realizar mediante la detección cambios bruscos en las intensidades de los píxeles, como puede ser la detección de bordes, líneas o puntos aislados, o la similitud de intensidades, como es el caso de la umbralización.

La umbralización consiste en calcular un valor, umbral o *threshold* en inglés, que vamos a utilizar para clasificar los píxeles en dos conjuntos. Un píxel pertenece a cada uno de estos conjuntos dependiendo de si su intensidad es mayor o menor que este valor. Llamaremos *fondo* al subconjunto de píxeles con intensidad menor o igual que el umbral y *objeto* a los píxeles con intensidad mayor que el umbral.

Podemos hablar de umbralización global, si se calcula un único umbral para toda la imagen, o local o adaptativa, si se usan distintos umbrales para distintas regiones de la imagen.

1.4.2 Verificación formal

Cuando queremos comprobar la corrección de un algoritmo podemos recurrir al uso de tests. Esta metodología solo nos permite comprobar una serie limitada de situaciones y valores de entrada, lo que supone que no podremos asegurar la corrección total del algoritmo.

Una solución a este problema es hacer uso de la especificación formal. Es decir, describir formalmente qué es lo que hace nuestro algoritmo haciendo uso de la lógica de predicados. Para hacer esto hay que proponer una precondición y postcondición. La precondición establece las condiciones que deben cumplir los parámetros de entrada de la función. La postcondición establece las relaciones entre los parámetros de entrada y el resultado de la función.

Cuando en los programas aparecen estructuras iterativas, es necesario especificar un invariante. El invariante es un predicado lógico que es cierto antes de entrar al bucle, durante la ejecución de este y después del bucle. Este invariante se utiliza posteriormente para verificar la postcondición.

Esto se puede hacer con distintos lenguajes. Uno de estos lenguajes es JML (*Java Modeling Languaje*) [3] que es utilizado para especificar código escrito en Java.

Algunas herramientas que permiten hacer verificación formal hacen uso de la lógica de Hoare. Ejemplos de estas herramientas son Krakatoa y KeY. Esta lógica provee una serie de axiomas y reglas de inferencia que nos permiten comprobar la corrección de los algoritmos.

2. Planificación

El desarrollo del trabajo ha seguido una planificación dividida en tareas realizadas de forma consecutiva. Las tareas se pueden ver detalladas en el siguiente esquema de descomposición de trabajo (EDT).

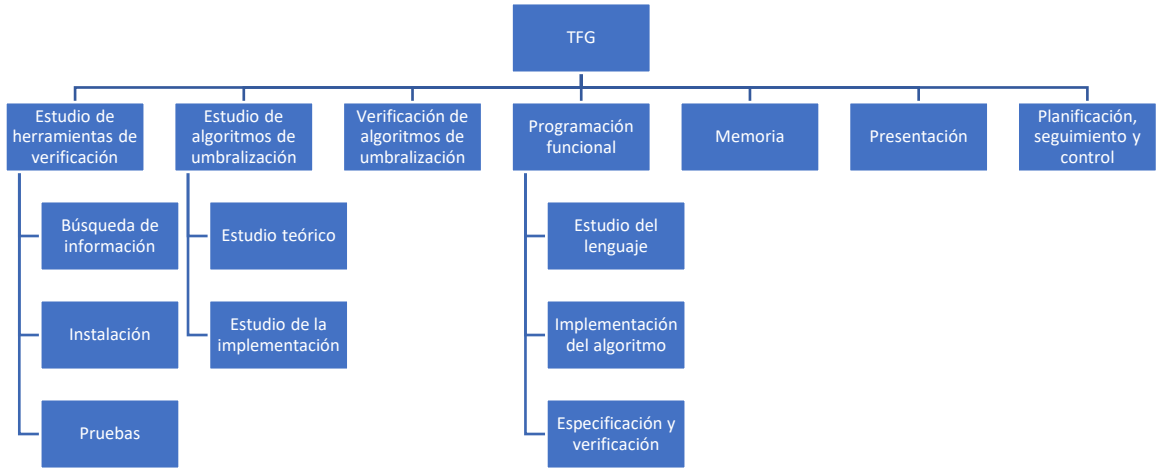


Ilustración 1: Diagrama de Descomposición de Trabajo

2.1 Explicación de las tareas

El trabajo se ha dividido en tareas principales subdivididas en tareas más simples. Su organización se puede ver en la ilustración anterior.

Búsqueda de información de herramientas de verificación formal (T-1.1)

Se hará un estudio del estado del arte en la verificación formal de programas implementados en java y se estudiarán sus características.

Instalación de las herramientas de verificación (T-1.2)

Se instalarán las herramientas seleccionadas para la realización de las pruebas.

Pruebas de las herramientas de verificación (T-1.3)

Se realizará una batería de pruebas contra código java especificado con JML.

Estudio teórico de algoritmos de umbralización (T-2.1)

En este apartado se hará un estudio de los algoritmos para conocer los conceptos matemáticos en los que se basan. Se evaluará la dificultad de verificación en función de su definición formal.

Estudio de la implementación de los algoritmos de umbralización (T-2.2)

En este apartado se estudiará la implementación propuesta y se evaluará su dificultad de verificación en función de la complejidad de su implementación.

Verificación formal de algoritmos de umbralización (T-3)

De los algoritmos estudiados se tratará de especificar y verificar formalmente una selección de ellos.

Estudio del lenguaje de programación funcional (T-4.1)

Estudio del lenguaje propuesto para la implementación de uno de los algoritmos.

Implementación de un algoritmo con paradigma funcional (T-4.2)

Implementación del algoritmo escogido en paradigma funcional.

Especificación y verificación del algoritmo (T-4.3)

Especificación y verificación del algoritmo implementado en paradigma funcional.

Memoria (T-5)

Redacción de la memoria del trabajo.

Presentación (T-6)

Realización de la presentación para la defensa del trabajo.

Planificación, Seguimiento y Control (T-7)

Planificación de la distribución de tiempos para las tareas y seguimiento y control de estas.

2.2 Planificación inicial

La planificación inicial fue calculada para depositar el trabajo entre el 25 y 27 de junio y realizar la defensa en julio.

La distribución de horas que se fijó fue la siguiente:

Tarea	Horas
T-1	20
T-2	20
T-3	150
T-4	50
T-5	30
T-6	20
T-7	10

Ilustración 2: Asignación de horas a cada tarea

La planificación temporal se estableció como muestra la figura.

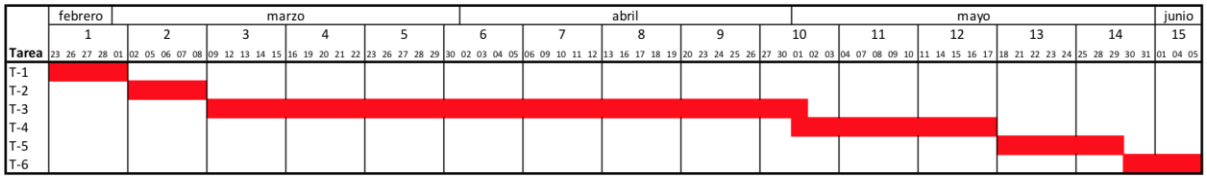


Ilustración 3: Distribución temporal de las tareas

2.3 Planificación modificada

Debido a un imprevisto, la dedicación de trabajo para el periodo del 26 de marzo al 8 de abril fue prácticamente nula por lo que se retrasó toda la planificación. Puesto que la fecha inicial de entrega era del 25 al 27 de junio se contaba con un margen de casi 3 semanas por lo que no supuso un problema para cumplir el objetivo de entrega y se seguía contando con una semana aproximada de margen.

Debido a que uno de los algoritmos, en concreto el algoritmo *isodata,* había requerido más tiempo del previsto, decidimos quitar las tareas del bloque *T-4.* Después añadimos una nueva tarea en la que se implementarían y especificarían una serie de métodos que nos permitieran hacer uso de los algoritmos de umbralización para binarizar imágenes. A esta tarea la denominaremos T-*8.*

Finalmente, tras encontrar un error en la verificación de un método y a problemas de índole personal y tras debatirlo con los tutores acordamos retrasar el depósito del TFG a la tercera convocatoria.

2.4 Seguimiento y control

En la siguiente tabla se mostrará la dedicación horaria planificada, la dedicación real y la desviación de los tiempos.

Tarea	Tiempo previsto	Tiempo Real	Desviación
T-1	20	20	0
T-2	20	25	+5
T-3	150	175	+25
T-4	50	0	-50
T-5	30	35	+5
T-6	20	X	X
T-7	10	10	0
T-8	0	30	+30
Total	**300**	**285+X**	

Ilustración 4: Comparativa tiempo previsto y tiempo real

3. Análisis

3.1 Estudio de herramientas de verificación de programas Java

Tras una búsqueda inicial en internet sobre este tipo de herramientas, las que más apariciones tienen son KeY Project, Krakatoa and Jessie y Open JML.

3.1.1 Krakatoa

Why [4] es una plataforma de verificación formada por 3 partes. La primera es un generador de condiciones de verificación de propósito general (VCG) llamado Why. Esta herramienta es utilizada como *back-end* de otras herramientas de verificación, pero también puede usarse directamente para la verificación de programas. Además, puede ser integrada con otras herramientas de verificación tanto interactivas, como Coq o Isabelle/HOL, o automáticas, como Alt-Ergo o CVC3.

La segunda herramienta es Frama-C utilizado para la verificación de programas escritos en C.

La última herramienta, y la que nos interesa en este caso, es Krakatoa [5] . Krakatoa es utilizado para la verificación de programas escritos en Java.

Los programas Java son anotados haciendo uso de una variante de JML. JML permite especificar el comportamiento de los módulos Java. Con la información aportada por las anotaciones se traduce el código a Jessie. Jessie es un lenguaje intermedio común a Java y a C. De este lenguaje se obtienen las condiciones de verificación que se utilizarán por los demostradores automáticos para verificar el código.

Las relaciones entre los distintos componentes pueden verse en la siguiente imagen.

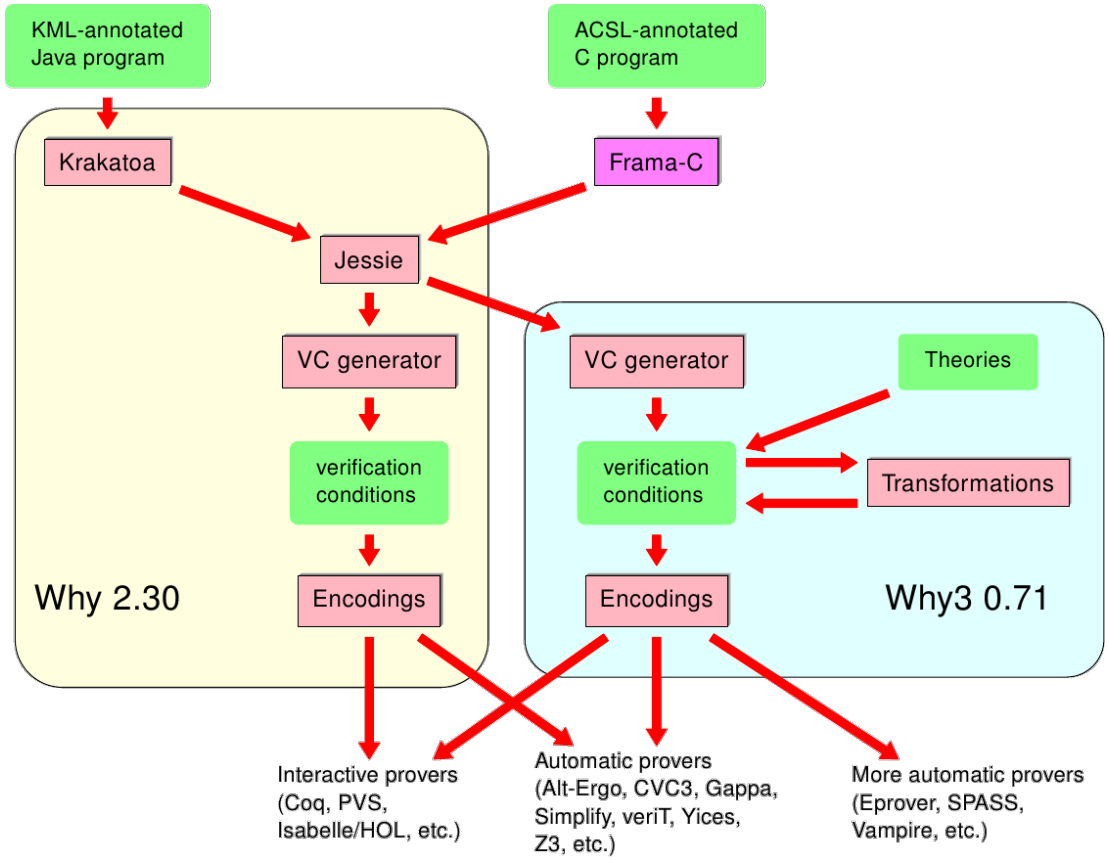


Ilustración 5: Relaciones entre los distintos componentes de Why. Imagen obtenida de [5]

Why ha dejado de desarrollarse y ha tomado el relevo Why3. Why3 no cuenta con un *front-end* propio como lo son Krakatoa o Frama-C. Por esta razón Jessie permite generar código intermedio para el generador de condiciones de Why3.

Instalación

Para la instalación de Krakatoa hemos utilizado una máquina virtual con un sistema operativo Ubuntu. Lo primero que haremos será descargar Why3 y Why2. Las versiones concretas han sido la 0.88.1 de Why3 y 2.39 para Why2. Realmente descargamos los ficheros fuente de Why que compilaremos más adelante con Ocaml. También instalaremos la librería Liblablgtk2 para la interfaz gráfica. Además de todo esto, necesario para compilar y ejecutar Why, necesitaremos instalar, al menos, un demostrador. Para hacer las pruebas he elegido Alt-Ergo. He utilizado la versión 0.99.1.

Las instrucciones de instalación se encuentran en el anexo.

Pruebas

Las pruebas que vamos a realizar tienen la finalidad de valorar las siguientes cuestiones:

1. Facilidad de uso
2. Información de errores
3. Funcionamiento de las anotaciones
4. Documentación
5. Capacidad para probar los algoritmos propuestos

Para la realización de las pruebas he utilizado algunos de los ejercicios de la práctica 4 de la asignatura *Especificación y desarrollo de sistemas de software.* Estos ejercicios implementan y especifican los siguientes algoritmos:

1. Sumar 3 a un entero
2. Calcular el máximo de dos enteros
3. Intercambiar dos elementos de un vector
4. Sumar dos enteros positivos (Versión sencilla)
5. Sumar dos enteros positivos (Versión *complicada*)
6. Calcular la raíz cuadrada
7. Calcular el máximo de un vector
8. Búsqueda del menor número en un vector (*Quickselect*)
9. Ordenar un vector de forma creciente

El código utilizado sirve de ejemplo para el uso de Krakatoa en la asignatura antes mencionada por lo que no ha sido necesario realizar modificaciones sobre las anotaciones. A continuación, se muestra uno de estos ejercicios. En concreto se trata de la versión *complicada* de la suma de dos enteros positivos. Se pueden apreciar las anotaciones JML en las que se establece la precondición, *requires*, la postcondición, *ensures,* el invariante, *loop_invariant,* y el variante, *loop_variant*.

```
1  /*@ requires (x>=0) && (y>=0);
2      @ ensures \result==x+y;
3      @*/
4    public static int suma2positivosComplicado (int x, int y){
5      int i=0, suma=x;
6      /*@ loop_invariant (0<=i<=y)&& (suma==x+i);
7        @ loop_variant y-i;
8        @*/
9      while (i<y){
10       suma=suma+1;
11       i++;
12     }
13     return suma;
14   }
```

Ilustración 6: Código del ejercicio sumar dos enteros positivos (versión complicada)

Usamos el comando *Krakatoa pruebas.java* para arrancar el programa. Si todo ha salido bien se nos mostrará la siguiente interfaz. En la siguiente imagen solo se ven 7 de los 9 ejercicios utilizados en las pruebas puesto que los ejercicios 8 y 9 se encuentran en ficheros independientes.

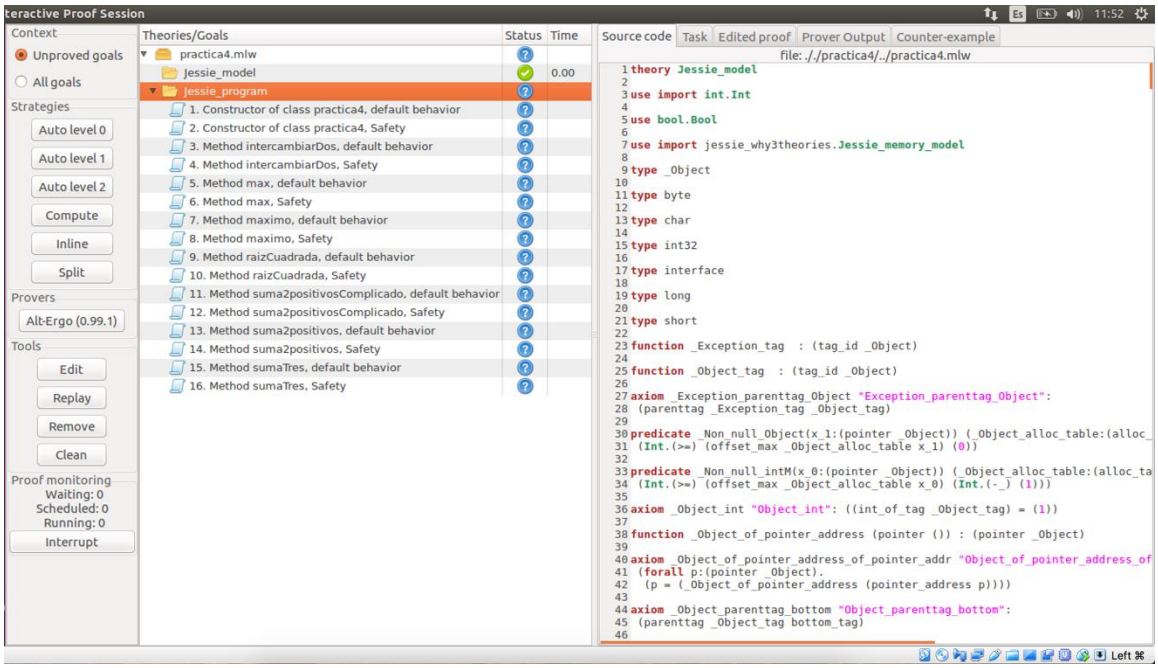


Ilustración 7: Interfaz de Why

La interfaz está dividida en tres paneles. En el primer panel de la izquierda podemos encontrar una serie de botones y selectores que nos van a permitir configurar de forma rápida los parámetros de la verificación. También nos muestra los diferentes demostradores que tenemos instalados. En el apartado *Strategies* aparecen distintas estrategias que podemos utilizar para simplificar la verificación. Estas permiten crear pasos intermedios en la verificación y cada uno de estos pasos puede ser probado por demostradores diferentes.

El panel central contiene las condiciones de verificación que los demostradores tratarán de verificar, y después de ejecutar los demostradores, el estado de cada una de ellas para cada uno de los demostradores y el tiempo y pasos que han tardado en demostrarse. Si en la columna *status* se muestra un símbolo verde se habrá conseguido probar la condición. En cualquier otro caso no se habrá conseguido.

El panel de la derecha contiene varias pestañas que nos muestran el código fuente y las tareas que tendrá que hacer el demostrador para tratar de verificar las condiciones.

Tras realizar la ejecución podemos observar, como era esperado, que el programa es capaz de procesar el código sin problemas. La segunda observación que podemos hacer es que el demostrador no ha sido capaz de demostrar todas las condiciones. En relación con esto, si hubiéramos tenido instalados más demostradores es posible que se hubieran podido demostrar, ya que en la asignatura que los utiliza se sabe que se consiguen demostrar todos.

Para realizar un análisis más complejo haremos pruebas con otros dos ficheros que contienen métodos más complejos con bucles anidados y declaración de axiomas y predicados para ayudar al demostrador a realizar su tarea. Estos dos ficheros son *FindNthLoweatNumber.java* y *Sort.java* que han sido facilitados por los tutores junto al código de prueba. Los resultados obtenidos son idénticos a los conseguidos con el primer fichero. No aparecen problemas al procesar las anotaciones y se consiguen demostrar los algoritmos.

Conclusiones

Krakatoa es sencillo de instalar si seguimos las instrucciones y también ofrece la oportunidad de utilizarlo como un plugin de Eclipse, aunque en este caso no ha sido utilizado. La respuesta del programa es muy visual y no se limita a mostrar salidas de consola. Esto facilita la tarea de corregir los errores y analizar las estadísticas del programa para cada uno de los pasos intermedios de la verificación.

3.1.2 KeY Project

KeY [6] es un proyecto de investigación a largo plazo iniciado en 1998. Su objetivo era el de la integración del análisis formal y la verificación formal en el desarrollo del software. Desde entonces se ha tomado como base para el desarrollo de herramientas de generación de test, herramientas de depuración y herramientas para enseñar la lógica de Hoare.

Al igual que Krakatoa utiliza un subconjunto de JML para anotar el código a verificar. También se puede usar *Java Dynamic Logic* para la especificación del código. KeY Project tiene algunas limitaciones en cuanto a las especificaciones JML soportadas.

Instalación

KeY se puede instalar como un plugin de Eclipse o de forma *standalone* en equipos con capacidad para ejecutar aplicaciones Java. Para realizar la instalación he seguido las instrucciones del sitio web [7] .

Pruebas

Los ficheros para realizar las pruebas serán los mismos que en el caso anterior. Puesto que KeY hace uso de un subconjunto de anotaciones JML diferente a Krakatoa tendré que realizar modificaciones que serán detalladas a continuación.

Tras probar los ficheros sin realizar modificaciones, el entorno de desarrollo nos advierte de que existen errores en las anotaciones JML. En concreto para el método *max(int[]v)* nos muestra un error en el que se nos informa de que no está soportada la palabra clave “loop-variant”. Para solucionar este error, y tras observar la documentación sobre JML, cambiamos la palabra clave “loop-variant” por “decreasing”. Realizamos el cambio para todas las apariciones de “loop-variant” para subsanar los errores.

Durante el uso normal del plugin, principalmente al guardar cambios, se muestra un error desde el IDE. El error es *java.lang.NullPointerException.* La interfaz no muestra más detalles del error por lo que no podemos deducir a qué se debe el error.

Después de eliminar todos los errores de las anotaciones JML usamos el plugin para realizar las pruebas. Al abrir las obligaciones de prueba para realizar la verificación salta una excepción *SLTranslationException* y nos advierte de que no se encuentra el tipo integer. Tras investigar encuentro el error en las anotaciones que definen los intervalos de la forma $x \geq y \geq z$. En su lugar hay que hacerlo de la forma $(x \geq y) \wedge (y \geq z)$.

Una vez corregidos todos los errores pasamos a tratar de verificar los algoritmos. KeY ha sido capaz de verificar todos los métodos de *pruebas.java* salvo el método de la raíz cuadrada. Cuando es incapaz de verificarlo de forma automática nos permite "ayudarle" de forma manual. A pesar de que nos ofrezca esta opción no vamos a dedicarle tiempo y nos centraremos en la verificación automática.

En el fichero Sort.java salen varios errores. En primer lugar, aparece que no soporta las palabras clave "for sorted", "for permutation" y "Permut {".

Después de quitar esas anotaciones y guardar aparecen nuevos errores.

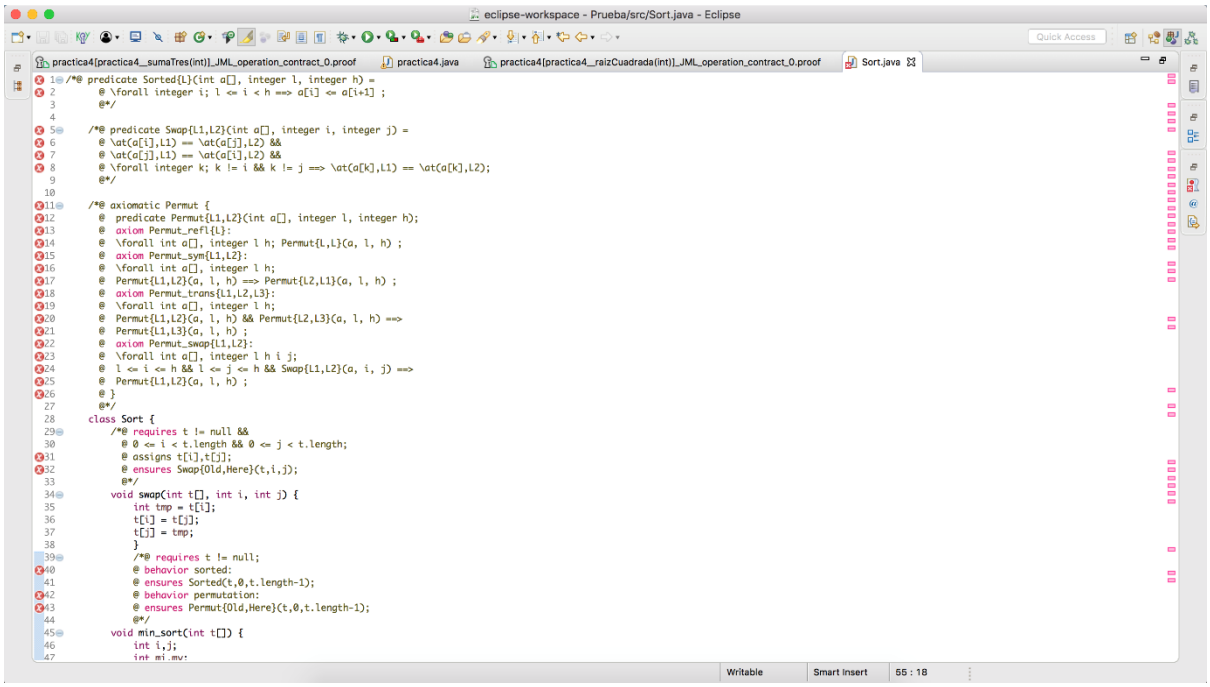


Ilustración 8: Errores en las anotaciones de KeY

KeY tampoco soporta la palabra clave "*predicate*". Para hacer esto en KeY es necesario hacer uso de los *taclets*. Los *taclets* permiten definir nuevas reglas lógicas, pero no funcionan exactamente igual que los predicados en Krakatoa [8].

Conclusiones

Los errores en las anotaciones eran de esperar debido al uso de otro subconjunto de JML. El plugin facilita la depuración de errores JML ya que señaliza de forma visual dónde se está produciendo el error. A pesar de esto, tiene algunos bugs que pueden dificultar la tarea de verificación. Además, cuenta con una serie de herramientas que para casos más complejos pueden resultar muy útiles como es el depurador gráfico.

Otro punto positivo es la instalación ya que el formato de *plugin* de Eclipse reduce su instalación al uso del asistente de Eclipse.

Algo que desfavorece esta herramienta es su complejidad a la hora de escribir predicados ya que no se utiliza el mismo formato que el utilizado en Krakatoa.

Debido a la experiencia previa en el subconjunto de JML utilizado por Krakatoa y a que lo único que en nuestro planteamiento aporta es la facilidad de detección de errores, Krakatoa resulta más indicada para nuestro trabajo que KeY.

3.1.3 OpenJML

OpenJML [9] es un conjunto de herramientas creado para reemplazar a ESC/Java2. No se trata de un demostrador de teoremas o comprobador de modelos. Estas herramientas trabajan transformando (traduciendo) JML en SMT-LIB para que posteriormente el *backend SMT solver* haga la verificación.

Instalación

OpenJML está disponible como un programa CLI (Interfaz de Línea de Comandos), como un plugin para Eclipse o como una aplicación web.

La instalación de las tres opciones se hace siguiendo las instrucciones de su página web [9].

Pruebas

En primer lugar, haremos una prueba con una instalación en Ubuntu. En la página se informa de que los verificadores no han sido probados en Ubuntu por lo que es posible que no funcionen.

Al tratar de hacer la prueba, la consola nos devuelve un error de sintaxis. Algunos de los errores que aparecían eran similares a los de KeY.

Uno de los errores nos indica que las variables especificadas como *"\bigint"* no pueden ser utilizadas como índice de un vector. Tras investigar los ejemplos me doy cuenta de que los tipos de las variables se declaran sin la contra barra.

Después de eliminar todos los errores de sintaxis aparece el siguiente error:

Uses uchecked or unsafe operations.

El error parece ser de Java y no de las anotaciones JML. Ejecuto el programa probando método a método para tratar de averiguar el origen del fallo. El error ocurre siempre por lo que investigo para averiguar si se trata de un error de la implementación o de JML. Finalmente averigüé que se trata de un error en la implementación. Lo más probable es que esté causado por un uso indebido de los genéricos de Java.

La segunda tanda de pruebas las realizo con el plugin para Eclipse.

Al tratar de ejecutar las pruebas con el mismo fichero, se produce el mismo error que en el caso anterior.

Finalmente realizo las pruebas con la versión Online [10].

OpenJML nos permite realizar pruebas desde su sitio web. En él introducimos nuestro código y el sitio se encarga de hacer peticiones contra un servidor donde se realizan las transformaciones para alimentar los demostradores. En primer lugar, realizaremos las pruebas con el fichero *pruebas.java*.

Lo primero que detecta son los errores de sintaxis. El error es el mismo que en el caso de KeY.

Ilustración 9: Error mostrado en la interfaz online de OpenJML

Tras eliminar los errores de sintaxis vuelvo a realizar la prueba y el sitio muestra el siguiente error:

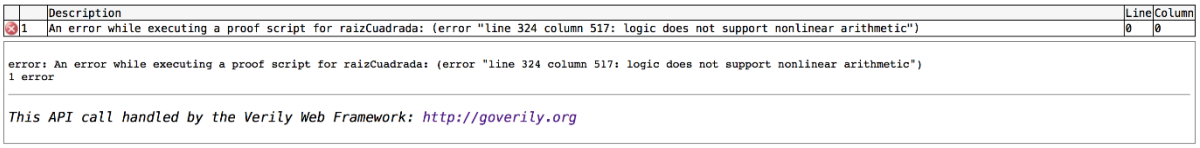


Ilustración 10: Error nonlinear arithmetic

El primer error que nos indica es que no es capaz de utilizar aritmética no lineal. Esta solo es utilizada en el método para calcular la raíz cuadrada por lo que simplemente la eliminamos del código y tratamos de verificar el resto del código. Tras eliminar el método el servicio consigue verificar el código.

Ahora trataremos de verificar el resto de los ficheros. En ambos casos el resultado que nos da el programa es el siguiente:

Ilustración 11: Error de posible bug

Nos informa de que se ha producido un error al tratar de verificar el programa o se trata de un bug de OpenJML. Nos piden que enviemos el código que estamos tratando de verificar al correo de los desarrolladores para que puedan investigar lo ocurrido.

Conclusiones

En general OpenJML funciona bastante mal si queremos hacer uso de él en sistemas Linux o como plugin de Eclipse. Para algoritmos complejos el sistema está bastante limitado ya que no se puede hacer uso de aritmética no lineal.

La versión online del mismo funciona bastante bien, pero tiene limitaciones. Una de estas limitaciones es que solo se puede hacer uso de un demostrador. Como parte positiva, cuenta con un buen sistema de *feedback* de errores sintácticos que facilitan la depuración del código que estamos tratando de verificar.

Debido a la cantidad de limitaciones que tiene frente a Krakatoa, he descartado el uso de esta herramienta.

3.2 Descripción de la herramienta elegida

La herramienta que he elegido para desarrollar el trabajo ha sido Krakatoa. Esta decisión se ha basado en primer lugar en lo completo de esta herramienta. Cuenta con una interfaz gráfica en la que se muestran los distintos algoritmos y pruebas que se generan a partir de las anotaciones del código. Si bien no cuenta con un sistema de verificación interactivo como KeY, sí que permite su integración con otras herramientas como Isabelle.

También ha tenido peso la experiencia previa en la herramienta que elimina la necesidad de dedicar tiempo a aprender un nuevo subconjunto de JML. Con KeY u OpenJML esto se tendría que haber hecho eliminando tiempo para otras partes del trabajo.

La amplia documentación existente, la comunidad y disponibilidad de ejemplos pueden facilitar el trabajo con Krakatoa.

3.2.1 Especificación con Krakatoa

Las anotaciones en Krakatoa se hacen utilizando una variante de JML. Para realizar las anotaciones hacemos uso de comentarios, bien sean comentarios en línea o comentarios en bloque. Para distinguir los comentarios normales de las anotaciones se hace uso del símbolo *@*. Después se hace uso de palabras clave para referirse a las diferentes características [11].

Para definir la precondición debemos hacer uso de la palabra clave *requires*. Para definir la postcondición debemos usar *ensures*. Cuando queremos acceder al valor de una variable de entrada que ha sufrido mutaciones, debemos utilizar *old* seguido del nombre de la variable.

Aparte de la precondición y de la postcondición también debemos especificar el invariante y el variante, si es que en nuestro método aparece algún bucle. El variante es una expresión que da como resultado un valor no negativo que decrece en cada iteración y que por tanto en algún momento se hace 0. El variante permite verificar que el bucle va a finalizar en algún momento. Para establecer el invariante se utiliza la palabra clave *loop_invariant* y para el variante *loop_variant.*

Para acceder a elementos de un vector que han sido modificados hacemos uso de *\at(array[componente],Pre).*

Es posible poder llevar un control sobre ciertos valores y variables para por ejemplo comprobar que no mutan durante la ejecución del programa haciendo uso de las variables fantasma. Para definir una variable fantasma debemos usar la estructura *ghost tipo variable = valor.* Cada vez que queramos hacer una modifiación de una variable fantasma debemos hacer uso de la palabra clave *ghost.* Sin embargo, para acceder al valor de la variable no es necesario y se pone su nombre directamente.

En algunos puntos concretos del código nos interesará verificar que se cumple una condición determinada. Para ello hacemos uso de los *asserts.* Para utilizarlos solo tenemos que añadir la siguiente estructura en el punto deseado: *assert expresión_lógica;.*

Para especificar nuevas funcionalidades y reglas podemos hacer uso de predicados, funciones lógicas, bloques axiomáticos y lemas.

Para especificar un predicado deberemos hacer uso de la siguiente estructura: *predicate nombre{etiqueta_estado} (tipo nombre_de_variable,…) = reglas;* Los predicados definidos van a devolver un valor booleano y pueden ser utilizados en cualquier punto de la especificación.

```
1  /*@ predicate isGrayScaleImage {L}(int I[]) =
2    @ (\forall integer i; (0<=i<I.length) ==> (0<=I[i]<=255));
3    @*/
```

Ilustración 12: Ejemplo de predicado

Las funciones lógicas nos permiten devolver valores numéricos. Para su definición hacemos uso de la estructura: *logic tipo_devuelto nombre{etiqueta_estado}(tipo nombre_de_variable,…) = reglas;*

Para establecer funcionalidades más complejas hacemos uso de los bloques axiomáticos. Los bloques se declaran de la siguiente manera:

axiomatic nombre {

[predicado|función_lógica];

axiom nombre_axioma{etiqueta de estado} :

reglas;

…

}

```
1  /*@ axiomatic isMaxIntensity {
2    @   predicate isMaxIntensity {L}(int v[], integer a);
3    @   axiom isMaxIntensity1{L}:
4    @     \forall int v[], integer a, integer i; (((a<i<v.length)&&(0<=a<v.length)) ==> ((v[i]==0)&&(v[a]>0)))==>isMaxIntensity{L}(v,a);
5    @   axiom isMaxIntensity1{L}:
6    @     \forall int v[], integer a, integer i;(((a==0)&&(v[i]==0)))==>isMaxIntensity{L}(v,a);
7    @ }
8    @*/
```

Ilustración 13: Ejemplo de predicado definido con un bloque axiomático

Para utilizar tanto las funciones lógicas como los predicados solo es necesario usar su nombre y completar la cabecera con las variables adecuadas.

Otra herramienta que podemos utilizar son los lemas. Los lemas nos permiten definir propiedades que se deben cumplir en la verificación de nuestros programas. Para definir un lema usamos la estructura *lemma nombre {etiqueta}: reglas;*

Para demostrar la corrección de los algoritmos, los demostradores hacen uso de las pruebas anteriores tomándolas como ciertas incluso cuando no han podido ser probadas. Cuando un lema no es demostrado, Krakatoa puede hacer uso de él para tratar de demostrar las pruebas. A diferencia de lo que ocurre con las pruebas, donde se ve claramente que se está haciendo uso de las pruebas anteriores, los lemas al estar en otra sección no se ven a primera vista y podemos suponer que la verificación es correcta aun cuando el lema sea que está utilizándose es falso.

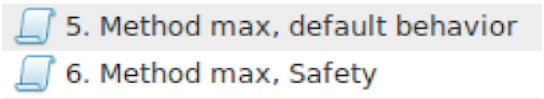
Al principio del programa podemos hacer uso de una serie de anotaciones como puede ser *//@+ CheckArithOverflow = no.* En esta anotación lo que le estamos diciendo a Krakatoa es que no tenga en cuenta los desbordamientos aritméticos.

Krakatoa genera dos tipos de pruebas a partir de las anotaciones JML. La primera son comprobaciones de seguridad, llamadas *safety,* donde se comprueban las siguientes situaciones:

1. División por cero
2. Desbordamiento aritmético
3. Referencias nulas
4. Accesos erróneos a índices de vectores

Como podemos ver lo que se está haciendo es comprobar que no se van a producir determinadas excepciones en tiempo de ejecución.

Por otro lado, se generan las pruebas necesarias para comprobar la corrección de los programas. A estas se les denomina *default behaviour.*



3.2.2 Limitaciones

Como hemos visto, Krakatoa hace uso de un subconjunto de JML para hacer la anotación y especificación del código. Debido a esta razón hay algunas características de JML que no están soportadas por Krakatoa. Algunas de estas características no soportadas son \sum, \product, \max, \min. Precisamente estas son características que nos hubieran sido muy útiles para especificar nuestros métodos y algoritmos. Para poder paliar esta situación he tenido que crear mi propio sistema para hacer sus funciones.

3.3 Estudio de algoritmos de umbralización

La umbralización [12] consiste en la obtención de un valor óptimo q que nos permita binarizar una imagen I. A partir de ese valor podemos construir dos subconjuntos con los píxeles de la imagen. Estos subconjuntos son el "Fondo" y el "Objeto". A cada píxel lo podremos clasificar únicamente como perteneciente a uno de estos dos conjuntos en función de si la intensidad del píxel es menor o igual que q o mayor que éste. Así queda

$$(u,v) \in \begin{cases} C_0 \; si \; I(u,v) \leq q \; (fondo) \\ C_1 si \; I(u,v) > q \; (objeto) \end{cases}$$

En la anterior ecuación u, v representa un píxel y $I(u,v)$ representa la intensidad del píxel cuyo valor se representa entre 0 y 255 donde 0 es negro y 255 blanco.

Una imagen puede ser representada por su histograma. El histograma representa la distribución de intensidades de una imagen indicando cuántos píxeles tienen una determinada intensidad.

Los algoritmos de umbralización pueden basarse en el uso del histograma de la imagen para calcular el valor q. Estos se pueden basar en la forma del histograma o en estadísticas. Los algoritmos que vamos a estudiar hacen uso de la información que se puede obtener de las dos formas anteriores.

3.3.1 Estadísticas obtenibles de un histograma

A continuación, explicaremos algunas estadísticas que van a aparecer en los algoritmos estudiados y su notación.

Número de píxeles pertenecientes al fondo y al objeto

$$n_0(q) = |n_0| = \sum_{g=0}^{q} h(g)$$

$$n_1(q) = |n_1| = \sum_{g=q+1}^{k-1} h(g)$$

$$MN = n_0(q) + n_1(q) = |C_0| + |C_1| = |C_0 \cup C_1|$$

donde $h(g)$ es el número de píxeles con intensidad g en la imagen, K es la intensidad máxima 256 y MN representa el total de píxeles de una imagen.

Media $\forall q$ de C_0 y C_1 y media de la imagen

$$\mu_0(q) = \frac{1}{n_0(q)} \cdot \sum_{g=0}^{q} g \cdot h(g)$$

$$\mu_1(q) = \frac{1}{n_1(q)} \cdot \sum_{g=q+1}^{k-1} g \cdot h(g)$$

$$\mu_I(q) = \frac{1}{MN} \cdot [n_0(q) \cdot \mu_0(q) + n_1(q) \cdot \mu_1(q)] = \mu_0(K-1)$$

Varianza $\forall q$ de C_0 y C_1 y varianza de la imagen

$${\sigma_0}^2(q) = \frac{1}{n_0(q)} \cdot \sum_{g=0}^{q} (g - \mu_0(q))^2 \cdot h(g)$$

$${\sigma_1}^2(q) = \frac{1}{n_1(q)} \cdot \sum_{g=q+1}^{k-1} (g - \mu_1(q))^2 \cdot h(g)$$

$$\sigma^2 = \frac{1}{MN} \cdot \sum_{g=0}^{K-1} (g - \mu_I) \cdot h(g) = \sigma_0^2(K-1)$$

$$\sigma^2 \neq \frac{1}{MN} [n_0(q) \cdot {\sigma_0}^2(q) + n_1(q) \cdot {\sigma_1}^2(q)]$$

3.3.2 Algoritmos globales

Los siguientes algoritmos hacen uso del histograma y las estadísticas anteriormente descritas. El valor q calculado es el mismo para todos los píxeles de la imagen.

3.3.2.1 Selección simple de umbral

Este método consiste en utilizar algunas de las estadísticas anteriores. Las estadísticas más utilizadas son la media, la mediana y el conocido como *MidRange* que es la media de la intensidad máxima y mínima.

Una generalización del caso de la mediana es el caso de los percentiles. Este método puede sernos de utilidad si conocemos la proporción aproximada de píxeles que van a pertenecer al fondo.

Estos algoritmos son muy rápidos y simples, pero tienen algunas desventajas:

- La utilización de la mediana supone que estamos asumiendo que la mitad de los píxeles de la imagen van a pertenecer al objeto y la otra mitad al fondo de la imagen. Esto no se ajusta a la mayoría de las situaciones y puede dar resultados no deseados. Por ejemplo, si deseamos obtener el texto escrito en un folio, la cantidad de píxeles pertenecientes al fondo va a ser mucho mayor que los pertenecientes al objeto.
- Si estamos utilizando la media, debemos tener en cuenta que es una estadística muy sensible a los valores extremos y puede hacer que la imagen binarizada no se ajuste a nuestras necesidades.

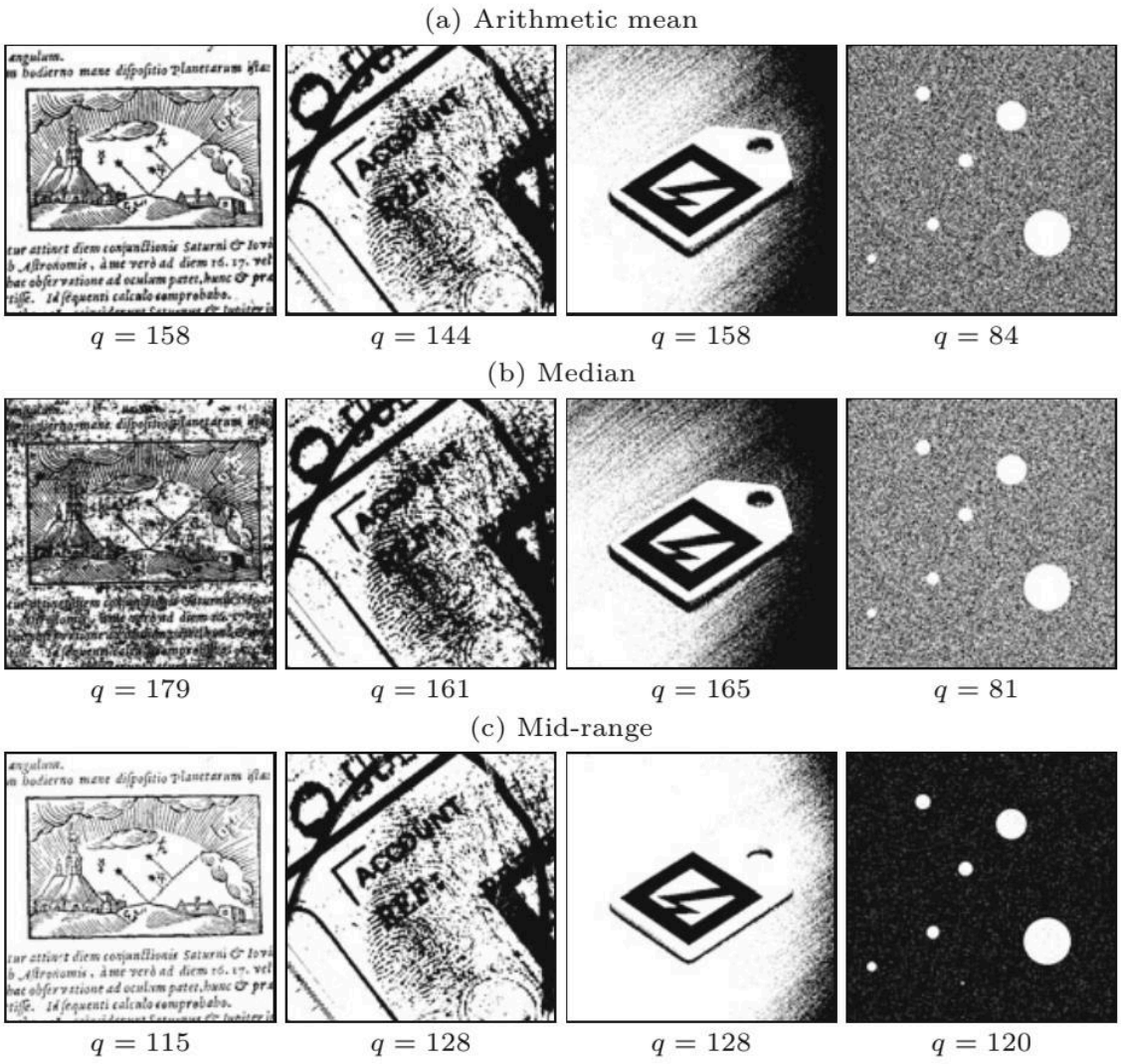


Ilustración 14 Imagen comparativa de algoritmos de umbralización. Imagen obtenida de [12]

3.3.2.2 Selección iterativa de umbral. (Isodata Threshold)

Este algoritmo parte de la suposición de que los píxeles de nuestra imagen siguen dos distribuciones Gaussianas diferentes. Además, las distribuciones tienen aproximadamente la misma varianza.

El algoritmo comienza tomando un valor aleatorio inicial, generalmente la media, como umbral y que divide los píxeles de la imagen en dos conjuntos. Una vez obtenidos los píxeles clasificados se calcula la media de cada uno de los conjuntos y se hace la media de los valores obtenidos. Ahora se tomará

ese valor como nuevo umbral y se repetirá el proceso hasta que el valor del threshold converja o se establezca un máximo de iteraciones.

Este algorimo se puede mejorar calculando las tablas de medias para cada posible valor q. De esta forma solo se recorre la imagen dos veces en vez de recorrerla en cada iteración del algoritmo. La complejidad del algoritmo es $\mathcal{O}(K)$.

3.3.2.3 Método de Otsu

Este método también supone que los píxeles pertenecen a dos clases distintas, pero en este caso la distribución que siguen es desconocida. El objetivo de este algoritmo es encontrar un valor q que dé como resultado dos distribuciones que cumplan:

- Que las distribuciones sean lo más estrechas posibles, es decir, que la varianza de cada una de ellas sea la menor posible.
- Las medias de las distribuciones estén lo más separadas posibles.

La anchura combinada de las dos distribuciones se pude calcular con las varianzas de las clases y se expresa con la ecuación

$$\sigma_w^2 = P_0(q) \cdot \sigma_0^2(q) + P_1(q) \cdot \sigma_1^2(q) = \frac{1}{MN} \cdot [n_0(q) \cdot \sigma_0^2(q) + n_1(q) \cdot \sigma_1^2(q)]$$

donde

$$P_0(q) = \sum_{i=0}^{q} p(i) = \frac{1}{MN} \cdot \sum_{i=0}^{q} h(i) = \frac{n_0(q)}{MN}$$

$$P_1(q) = \sum_{i=q+1}^{K-1} p(i) = \frac{1}{MN} \cdot \sum_{i=q+1}^{K-1} h(i) = \frac{n_1(q)}{MN}$$

son las respectivas probabilidades de las clases C_0 y C_1.

La distancia entre las medias se puede medir con la varianza entre clases y se expresa con la ecuación.

$$\sigma_b^2 = P_0(q) \cdot (\mu_0(q) - \mu_I)^2 + P_1(q) \cdot (\mu_1(q) - \mu_I)^2$$

$$= \frac{1}{MN} \cdot [n_0(q) \cdot (\mu_0(q) - \mu_I)^2 + n_1(q) \cdot (\mu_1(q) - \mu_I)^2]$$

Sabiendo que la varianza de la imagen es constante y que cumple $\sigma_I^2 = \sigma_w^2(q) + \sigma_b^2(q)$, para $q = 0, \dots, k-1$ podemos obtener nuestro valor q o bien maximizando σ_b^2 o bien minimizando σ_w^2. Lo más adecuado es maximizar σ_b^2 ya que solo hace uso de la media y es fácil de obtener con el histograma. La media de la imagen se puede calcular como la media ponderada de las medias de cada uno de los conjuntos y podemos simplificar la ecuación σ_b^2 a

$$\sigma_b^2 = P_0(q) \cdot P_1(q) \cdot [\mu_0(q) - \mu_1(q)]^2$$

$$= \frac{1}{(MN)^2} \cdot n_0(q) \cdot n_1(q) \cdot [\mu_0(q) - \mu_1(q)]^2$$

Al depender únicamente de las medias en vez de las varianzas la implementación del algoritmo es muy eficiente. Para ello se asume que la imagen tiene N píxeles y K intensidades. Para mejorar el

rendimiento se puede, al igual que en el algoritmo Isodata, precalcular los valores de las medias para cada posible valor q.

3.3.2.4 Umbral de máxima entropía

La entropía es un importante concepto de la teoría de la información y más en concreto de la compresión. La entropía mide el promedio de la información contenido en un mensaje generado por una fuente de información estocástica.

Podemos suponer los píxeles que forman una imagen I como un mensaje de MN símbolos que pertenecen a un alfabeto K. Este alfabeto estaría constituido por intensidades entre 0 y 255. Si asumimos que los píxeles son independientes y conocemos la probabilidad de que una intensidad g ocurra, entonces la entropía mide la probabilidad de obtener una imagen concreta.

Para definir la entropía de una imagen hace falta conocer el alfabeto $K = \{0,1, \dots, k-1\}$ y el vector de probabilidades o función de densidad $(p(0), p(1), \dots p(k-1))$. Podemos estimar la probabilidad a priori $\mathrm{p}(g)$ a través de la función de densidad de la imagen normalizando el histograma de la siguiente manera $p(g) \approx \mathrm{P}(g) = \frac{h(g)}{MN}$ para $0 \leq g \leq k$ tal que $0 \leq \mathrm{p}(g) \leq 1$ y $\sum_{g=0}^{k-1} \mathrm{p}(g)$. La función de distribución acumulada queda entonces de la siguiente manera

$$\mathrm{P}(g) = \sum_{i=0}^{g} \frac{h(g)}{MN} = \sum_{i=0}^{g} \mathrm{p}(i)$$

donde $\mathrm{P}(0) = \mathrm{p}(0)$ y $\mathrm{P}(k-1) = 1$.

Con la probabilidad de distribución estimada $p(g)$ podemos definir la entropía de la imagen como

$$H(Z) = \sum_{g \in Z} \mathrm{p}(g) \cdot \log_b \left(\frac{1}{\mathrm{p}(g)} \right) = -\sum_{g \in Z} \mathrm{p}(g) \cdot \log_b (\mathrm{p}(g))$$

donde $g = I(u,v)$ y $\log_b(x)$ es el logaritmo de x en base b. Si tomamos b como 2, entonces la entropía de la imagen estará medida en bits.

De entre los métodos que hacen uso de la entropía para calcular umbrales, vamos a estudiar la técnica de Kapur.

Dado un umbral particular q, las probabilidades de distribución estimadas para cada partición C_0 y C_1 vienen determinadas por

$$C_0 : \left(\frac{\mathrm{p}(0)}{\mathrm{P}_0(q)} \frac{\mathrm{p}(1)}{\mathrm{P}_0(q)} \dots \frac{\mathrm{p}(q)}{\mathrm{P}_0(q)} 0\ 0 \dots\ 0 \right)$$

$$C_1 : \left(0\ 0 \dots 0 \frac{\mathrm{p}(q+1)}{\mathrm{P}_1(q)} \frac{\mathrm{p}(q+2)}{\mathrm{P}_1(q)} \dots \frac{\mathrm{p}(k-1)}{\mathrm{P}_1(q)} \right)$$

Las probabilidades acumuladas son

$$\mathrm{P}_0(q) = \sum_{i=0}^{q} \mathrm{p(i)} = \mathrm{P}(q)\ y\ \mathrm{P}_1(q) = \sum_{i=q+1}^{K-1} \mathrm{p(i)} = 1 - \mathrm{P}(q)$$

Dado que el fondo y el objeto son dos particiones disjuntas $\mathrm{P}_0(q)$+$\mathrm{P}_1(q) = 1$. La entropía general de la imagen la podemos denotar como $H_{01}(q) = H_0(q) + H_1(q)$. La entropía de cada una de las

particiones la podemos describir de forma que resulte computacionalmente más eficiente de la siguiente manera

$$H_0(q) = -\frac{1}{\mathrm{P}_0(q)} \cdot S_0(q) + \log(\mathrm{P}_0(q))$$

$$H_1(q) = -\frac{1}{1-\mathrm{P}_0(q)} \cdot S_1(q) + \log(1-\mathrm{P}_0(q))$$

Y a partir de la probabilidad de distribución $\mathrm{p}(i)$ la probabilidad acumulada P_0 y las sumas S_0 y S_1 pueden ser calculadas de forma recursiva

$$\mathrm{P}_0(q) = \begin{cases} \mathrm{p}(0) & \text{para } q = 0, \\ \mathrm{P}_0(q-1) + \mathrm{p}(q) & \text{para } 0 < q < k, \end{cases}$$

$$\mathrm{S}_0(q) = \begin{cases} \mathrm{p}(0) \cdot \log\big(p(q)\big) & \text{para } q = 0, \\ \mathrm{S}_0(q-1) + \mathrm{p}(q) \cdot \log(\mathrm{p}(q)) & \text{para } 0 < q < k, \end{cases}$$

$$\mathrm{S}_1(q) = \begin{cases} 0 & \text{para } q = 0, \\ \mathrm{S}_0(q-1) + \mathrm{p}(q+1) \cdot \log(\mathrm{p}(q+1)) & \text{para } 0 < q < k. \end{cases}$$

La complejidad del algoritmo es $\mathcal{O}(K)$.

3.3.2.5 Umbral de mínimo error

Este algoritmo funciona buscando una mezcla de distribuciones Gaussianas que se ajusten al histograma de la imagen.

Debemos asumir que los píxeles de la imagen pertenecientes al objeto y fondo tienen una intensidad aleatoria obtenida de distribuciones Gaussianas con μ y σ^2 desconocidas. Para cada píxel tendremos que decidir a qué clase es más probable que pertenezcan. La probabilidad de observar una intensidad x en el fondo es $p(x|C_0)$ y la probabilidad de que aparezca en el objeto es $p(x|C_1)$. Vamos a suponer que conocemos estas intensidades.

Ahora queremos saber la probabilidad de que un píxel pertenezca al fondo o al objeto. Estas son las llamadas probabilidades a posteriori. Estas se denotan como $p(C_0|x)$ y $p(C_1|x)$. Para clasificar los píxeles solo tenemos que escoger la probabilidad que sea mayor.

Según el teorema de Bayes podemos estimar las probabilidades a posteriori de la siguiente manera:

$$p(C_j|x) = \frac{p(x|C_j) \cdot p(C_j)}{p(x)}$$

Ahora podemos formular la regla de decisión de la siguiente manera:

$$C = \begin{cases} C_0 & \text{si } p(x|C_0) \cdot p(C_0) > p(x|C_1) \cdot p(C_1), \\ C_1 & \text{en cualquier otro caso.} \end{cases}$$

Esta regla se llama regla de decisión de Bayes y minimiza la probabilidad de hacer una clasificación errónea. Antes hemos dicho que las distribuciones estaban modeladas como distribuciones Gaussianas así que podemos reformular las probabilidades a posteriori de la siguiente manera:

$$p(x|C_j) \cdot p(C_j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} \cdot exp\left(-\frac{(x-\mu_j)^2}{2\sigma_j^2}\right).p(C_j)$$

Aplicando logaritmos naturales a la anterior ecuación obtenemos la siguiente expresión que minimizándola nos permite nos permite encontrar la clase C_j que maximiza $p(x|C_j) \cdot p(C_j)$.

$$\varepsilon_j(x) = \frac{(x-\mu_j)^2}{2\sigma_j^2} + 2 \cdot \ln(\sigma_j) - \ln\left(p(C_j)\right)$$

$\varepsilon_j(x)$ se puede interpretar como el error potencial que podemos cometer al clasificar el valor x como perteneciente a la clase C_j. A partir de esto nuestra regla de decisión queda ahora como

$$C = \begin{cases} C_0 & \text{si } \varepsilon_0(x) \leq \varepsilon_1(x)\,, \\ C_1 & \text{en cualquier otro caso.} \end{cases}$$

Para un umbral q, podemos medir la bondad de la clasificación con la función

$$e(q) = \sum_{g=0}^{q} \mathrm{p}(g) \cdot \varepsilon_0(g) + \sum_{g=q+1}^{k-1} \mathrm{p}(g) \cdot \varepsilon_1(g)$$

Si ahora sustituimos $\varepsilon_1(g)$ en la anterior ecuación y hacemos algunas operaciones obtenemos

$$e(q) = 1 + P_0(q) \cdot \ln(\sigma_0^2(q)) + P_1(q) \cdot \ln(\sigma_1^2(q)) - 2 \cdot P_0(q) \cdot \ln(P_0(q)) - 2 \cdot P_1(q) \cdot \ln(P_1(q))$$

A partir del histograma podemos calcular las probabilidades a priori P_0 y P_1y las correspondientes varianzas σ_0 y σ_1. Estos valores pueden calcularse de la siguiente manera

$$P_0(q) = \frac{n_0(q)}{MN}$$

$$P_1(q) = \frac{n_1(q)}{MN}$$

$$\sigma_0^2 = \frac{1}{n_0(q)} \cdot \left[B_0(q) - \frac{1}{n_0(q)} \cdot A_0^2(q)\right]$$

$$\sigma_1^2 = \frac{1}{n_1(q)} \cdot \left[B_1(q) - \frac{1}{n_1(q)} \cdot A_1^2(q)\right]$$

donde

$$A_0 = \sum_{g=0}^{q} \mathrm{h}(g) \cdot g$$

$$B_0 = \sum_{g=0}^{q} \mathrm{h}(g) \cdot g^2$$

$$A_1 = \sum_{g=q+1}^{k-1} \mathrm{h}(g) \cdot g$$

$$B_1 = \sum_{g=q+1}^{k-1} \mathrm{h}(g) \cdot g^2$$

Estos valores se pueden calcular de forma recursiva con las siguientes relaciones

$$A_0(q) = \begin{cases} 0 & \text{para } q = 0, \\ A_0(q-1) + \mathrm{h}(q) \cdot q & \text{para } 1 \leq q \leq k-1, \end{cases}$$

$$B_0(q) = \begin{cases} 0 & \text{para } q = 0, \\ B_0(q-1) + \mathrm{h}(q) \cdot q^2 & \text{para } 1 \leq q \leq k-1, \end{cases}$$

$$A_1(q) = \begin{cases} 0 & \text{para } q = k-1, \\ A_1(q+1) + \mathrm{h}(q+1) \cdot (q+1) & \text{para } 0 \leq q \leq k-2, \end{cases}$$

$$B_1(q) = \begin{cases} 0 & \text{para } q = 0, \\ B_1(q+1) + \mathrm{h}(q+1) \cdot (q+1)^2 & \text{para } 0 \leq q \leq k-2. \end{cases}$$

Haciendo dos pasadas sobre el histograma se calculan los valores para todo q de σ_0^2 y σ_1^2. Con un tercer paso se puede calcular el valor q mínimo. La complejidad de este algoritmo es $\mathcal{O}(K)$.

3.3.3 Conclusiones

Como hemos podido observar, casi todos los algoritmos globales tienen una complejidad $\mathcal{O}(K)$ ya que están implementados haciendo uso de una estructura iterativa, en algún caso varias estructuras iterativas secuenciales y en ningún caso utilizan estructuras iterativas anidadas.

Para ello se hacen uso de tablas precalculadas para cada posible valor q y que además siguen una definición recursiva que hace uso de los valores ya calculados para no tener que recorrer nuevamente el histograma. La dificultad en la definición formal de estos algoritmos no es muy grande y nos permite dar un valor directo en función de las variables de entrada. Esta permite que la especificación formal no dependa de la implementación del algoritmo.

Dentro de estos hay una excepción que es el algoritmo de selección iterativa. Este sigue una definición recursiva que finaliza cuando se consigue un valor q estable. En este caso la definición será más compleja puesto que es necesario especificar la condición de parada del algoritmo. Aquí es imposible definir formalmente el valor a partir de los parámetros de entrada y tendremos que hacerlo de forma recursiva. Esta especificación va a ser dependiente de la implementación que se haga del algoritmo.

4. Especificación y verificación formal de los algoritmos

Para la implementación de los algoritmos he tomado como base la implementación realizada para el libro [12]. Los algoritmos están implementados a partir de una clase Thresholder que tiene todos los métodos auxiliares y un método abstracto Threshold(). Las distintas clases redefinen este método para implementar el algoritmo concreto. Para simplificar el proceso de verificación con Krakatoa he creado un único fichero con una clase en la que se implementarán los distintos algoritmos tratando de transcribirlos haciendo las mínimas modificaciones necesarias y sin que afecten a la estructura original del algoritmo.

Las imágenes con las que vamos a trabajar son imágenes en escala de grises de 8 bits. Esto supone que la intensidad de cada píxel puede oscilar entre 0, si es totalmente negro, o 255, si es totalmente blanco. De esta manera los histogramas usados serán representados por vectores de enteros con 256 componentes. Cada una de sus componentes representan las intensidades de 0 a 255 y su contenido el número de píxeles de la imagen que tienen esa intensidad. En la implementación serán denotados como $int\ h[]$.

Las imágenes serán representadas con vectores de enteros con longitud indeterminada y cuyas componentes deben estar entre 0 y 255. Las imágenes serán denotadas como $int\ I[]$.

Para la verificación he instalado dos demostradores más. Estos demostradores han sido *CVC3* en la versión 2.4.1 y *Z3* en la versión 4.4.1.

Los algoritmos de umbralización que he verificado han sido los siguientes:

- Selección simple.
    - Media
    - Mediana
    - Percentil
    - *MidRange*
- Iterativo (Isodata).

Algunos de estos algoritmos hacen uso de métodos auxiliares para implementar su funcionamiento. Para poder realizar la verificación de los algoritmos también he tenido que verificar estos métodos.

Los métodos auxiliares han sido:

- int countPixel(int v[])
- int count(int[] h, int lo, int hi)
- int minIntensityComponent(int[] h)
- int maxIntensityComponent(int[] h)
- int doubleToInt (double x)

También se han desarrollado otros métodos que finalmente no han sido utilizados en el desarrollo del trabajo. Su implementación se debió a una interpretación errónea del algoritmo *midRange.* A pesar de no haber resultado útiles en la implementación final del algoritmo, pueden ser útiles para otros trabajos. Estos han sido:

- int max (int[] v)
- int min (int[] v)
- int maxComponent(int[] v)
- int minComponent (int[] v)

Para poder especificar y verificar tanto los métodos auxiliares como los algoritmos también he hecho uso de funciones lógicas, predicados y lemas. Las funciones lógicas y algunos de los predicados han sido definidos mediante el uso de bloques axiomáticos.

Funciones lógicas

1. sum
2. sumHist
3. mean
4. doubleToInteger
5. count
6. isoData

Predicados

1. isPositiveArray
2. isGrayScaleImage
3. isMaxIntensity
4. isMinIntensity

Lemas

1. L1 propiedad división de enteros en desigualdad menor o igual que
2. L2 propiedad división de enteros en desigualdad menor que
3. L3 propiedad división de enteros en desigualdad menor o igual que

4.1 Funciones lógicas

4.1.1 sum

Esta función lógica va a reproducir la funcionalidad que en JML tiene la palabra clave *sum.* Está definida por un bloque axiomático que nos va a permitir sumar las componentes de una sección concreta de un vector de enteros. Su implementación está basada en los ejemplos de la galería *toccata* [13] y el código de la práctica 4 de la asignatura "Especificación y Desarrollo de Sistemas de Software".

El bloque axiomático define una función de tipo logic. Las funciones de tipo logic devuelven como resultado un tipo *char, byte, short, float* o *integer*. En nuestro caso nuestra función va a sumar las componentes entre las componentes *a* y *b* sin incluir la componente *b*. He decidido hacerlo así porque se puede hacer tanto la suma de todas las componentes de un vector, como de un rango concreto dentro del vector.

```
/*@ axiomatic Sum {
  @   logic integer sum{L}(int [] v, integer a, integer b);
  @   axiom sum1{L} :
  @   \forall int v [], integer a, integer b; b>=v.length ==>
  @     sum(v,a,b) == sum(v,a,v.length-1);
  @   axiom sum2{L} :
  @   \forall int v [], integer a, integer b; a<0 ==>
  @     sum(v,a,b) == sum(v,0,b);
  @   axiom sum3{L} :
  @     \forall int v [], integer a, integer b; b <= a ==> sum(v,a,b) == 0;
  @   axiom sum4{L} :
  @     \forall int v [], integer a, integer b; b > a ==>
  @       sum(v,a,b) == sum(v,a,b-1) + v[b-1];
  @ }
  @*/
```

Ilustración 15: Código de la función lógica Sum

La cabecera en nuestro caso es *logic integer sum{L}(int []v, integer a, integer b);*. La etiqueta *{L}* nos indica que se trata de una función lógica híbrida. Esto quiere decir que va a depender de estados de la memoria. Las etiquetas representan los estados de memoria. Después de haber definido la cabecera de la función hay que especificar los axiomas que definen el comportamiento de la función. En este caso lo he definido con cuatro axiomas. El primer axioma especifica que si *b* es mayor que el número de componentes del vector solo se realiza la suma hasta la última componente. El segundo especifica que si el parámetro *a* es menor que 0 se debe empezar a contar desde la componente 0. Estos dos axiomas evitan que los índices se salgan de los valores válidos. El tercer axioma es el que especifica el caso base de la recursividad. En este caso si el índice *b* alcanza al índice *a* el resultado que devuelve la función es 0. En el cuarto axioma se especifica la recursividad indicando que si *b* es mayor que *a* entonces lo que devuelve la función es el valor de *v[b-1]* más el resultado de llamar a la función *sum* con el parámetro *b* igual a *b-1*.

4.1.2 sumHist

Esta función lógica especifica, usando un bloque axiomático, la funcionalidad de una suma ponderada de un conjunto concreto de un vector de enteros. La cabecera de la función es *logic integer sumHist{L}(int [] h, integer a, integer b).* Supondremos que el vector representa un histograma donde $f_i$representa el valor de cada posición *i* del vector y el valor de cada componente f_i. Así lo que este método está calculando es $\sum_{i=a}^{b-1} i\, f_i$.

La especificación es prácticamente igual que la de *sum* ya que parte del mismo esquema y los mismos supuestos ante los casos raros. Estos son: que *b* sea menor que *a*, que *a* sea menor que 0 y que *b* sea mayor que la longitud del vector. Los cambios estarían en el axioma 4 que quedaría así: *\forall int h [], integer a, integer b; b > a ==> sumHist(h,a,b) == sumHist(h,a,b-1) + (h[b-1]*(b-1)).* De esta forma se calcularía cada sumando if_i.

4.1.3 mean

Otra función que necesitamos es la de la media. La cabecera de la función de este bloque es *logic double mean{L}(int [] h, integer a, integer b).* Se va a devolver un *double* con la media de los elementos entre *a* y *b*, ambos incluidos.

Este bloque es uno de los que más axiomas tiene ya que hay que tener en cuenta más casos especiales. Además de los dos casos comunes a *sum* y *sumHist* hay que tener en cuenta lo que ocurre si tratamos de calcular la media sobre un rango en el que no hay población, es decir, que la suma de las componentes sea 0. En ese caso estaríamos tratando de hacer una división 0/0 lo que produciría un

error. Esto lo arreglamos añadiendo el siguiente axioma: *\forall int h [], integer a, integer b; ((sum(h,a,b+1)==0) && (b<h.length)) ==> (mean(h,a,b) == 0)*. Así definimos que cuando se dé esa situación la media sea 0. Si *a* es mayor que *b* entonces el resultado es 0. A diferencia de los otros dos bloques axiomáticos, este no tiene caso de recursividad y para los casos normales el resultado que se devuelve es *sumHist(h,a,b+1)/sum(h,a,b+1).* En este caso se suma 1 al parámetro *b* ya que la media sí que tiene en cuenta la componente de índice *b* a diferencia de *sum* y *sumHist.*

4.1.4 doubleToInteger

Este bloque nos permitirá transformar un doble en un entero. Ha sido necesaria la definición de este método puesto que Krakatoa falla cuando se trata de hacer un cast directo.

```
1  /*@ axiomatic DoubleToInteger {
2    @   logic integer doubletoint{L}(real n);
3    @   axiom doubletoint1{L} :
4    @     \forall real n; \exists integer a; (n-1 < a <= n) && (doubletoint(n)==a);
5    @ }
6    @*/
```

Ilustración 16: Función lógica DoubleToInteger

Su cabecera es *logic integer doubletoint{L}(real n).* El axioma que calcula el resultado se especifica haciendo uso de un cuantificador existencial para definir que existe un entero *a*. Este entero cumple que, para todo real *n*, *a* es mayor que *n-1* y menor o igual que *n*. El *a* menor o igual que *n* en realidad supone que la diferencia *n-a* es menor que 1.

4.1.5 count

Otra funcionalidad que necesitamos es la de poder contar la frecuencia de aparición de un entero en un vector de enteros. Para ello usamos este bloque axiomático con la siguiente función lógica *logic integer count{L}(int [] v, integer a, integer b, integer x)*. Los parámetros representan el vector en el que realizar la búsqueda, el índice al partir del cual empezar a buscar, el índice hasta el que vamos a buscar y el entero cuya frecuencia de aparición queremos contar. Volvemos a contar con dos axiomas que controlan los casos raros en los que *a* es menor que 0 y *b* mayor que la longitud del vector. Tenemos el caso base que devuelve 0 que se corresponde a *b* menor o igual que *a* y dos casos particulares para la definición recursiva del método. El primero es el caso en el que el elemento actual contenga el elemento *x* que devolverá 1 más el resultado de llamar al método con los mismos datos, pero cambiando el parámetro *b* por *b-1*. El otro caso es en el que el elemento no se corresponde con la *x*. En este último caso directamente devolvemos mismo que en el anterior caso, pero sin sumarle 1 al resultado que obtengamos.

4.1.6 isoData

Este bloque va a recoger las condiciones que debe cumplir un umbral válido obtenido mediante el algoritmo isodata. La función lógica de este bloque es *logic integer isodata*{L}(int [] h, integer i, integer q, integer lim). Los parámetros se corresponden con el vector que representa el histograma al que se le va a aplicar el algoritmo, la longitud del vector, el umbral del que se parte y el número de iteraciones límite.

He tenido que definir el funcionamiento del algoritmo ya que es muy difícil especificar las propiedades que cumple un umbral calculado con el mismo. Dado que se trata de un algoritmo iterativo y que no hay garantías de que el algoritmo pueda converger rápidamente al resultado hacemos uso de un límite de iteraciones.

Además de ese límite de iteraciones también hay que tener en cuenta casos excepcionales que se pueden dar durante el resultado del algoritmo y que resultarían en la detención inmediata de la ejecución del algoritmo. Estas condiciones que se especifican en el axioma 1 del bloque consisten en que uno de los dos subconjuntos de píxeles creados a la izquierda y la derecha del umbral inicial se queden sin píxeles. En el caso de que esto ocurra se devuelve -1.

Otra de las condiciones de parada es que el método se estabilice y el nuevo umbral calculado se corresponda con el anterior. En este caso devolveremos el umbral que habíamos calculado.

Si el número de iteraciones es alcanzado, entonces se devolverá el umbral anterior calculado.

En caso de que no se hayan alcanzado las iteraciones máximas ni el algoritmo se haya estabilizado, se calculará la media de los centros de los dos subconjuntos y se utilizará para llamar de forma recursiva al método reduciendo en una unidad el límite de iteraciones y pasando el nuevo umbral como umbral inicial.

4.2 Predicados

4.2.1 isMaxIntensity

Este bloque, a diferencia de los anteriores, está definiendo un predicado complejo. Es decir, el resultado no va a ser un número, va a ser un booleano. El predicado es *predicate isMaxIntensity {L}(int v[], integer a)* y lo que hace es comprobar si la componente *a* se corresponde con la intensidad máxima del histograma representado por *v*. Para hacer esto deben cumplirse dos condiciones: que todas las componentes a la derecha de la posición *a* sean ceros y que *v[a]* sea distinto de cero. Estableciendo estas dos condiciones tendríamos que especificar también el caso en el que *v[a]* sea 0 ya que hemos establecido que en una imagen vacía la intensidad máxima es 0.

4.2.2 isMinIntensity

Este bloque es muy similar al anterior. El predicado es exactamente igual a diferencia del nombre *predicate isMinIntensity {L}(int v[], integer a).* La principal diferencia con el método anterior es el primer axioma donde se establece que todas las componentes a la derecha de la componente *a* tienen que ser 0.

4.2.3 isPositiveArray

Este predicado comprueba que todas las componentes de un vector sean mayores o iguales que 0. Si esto se cumple entonces el vector puede representar un histograma. Su cabecera es *predicate isPositiveArray {L}(int v[], integer a)* donde *v* es el vector a comprobar y *a* su longitud. La creación de este predicado tiene la finalidad de no repetir el código en todas las especificaciones de los métodos y algoritmos. En este caso se define directamente sin necesidad de usar bloques axiomáticos.

4.2.4 isGrayScaleImage

Este predicado comprueba que un vector representa una imagen en escala de grises codificada en 256 bits. La cabecera es *predicate isGrayScaleImage {L}(int I[])* donde *I* es el vector a comprobar. Si para toda componente del vector los valores están comprendidos entre 0 y 255, ambos incluidos, entonces el vector puede considerarse como una imagen en escala de grises.

4.3 Lemas

Los lemas escritos han sido utilizados para facilitar la tarea a los demostradores automáticos que hemos utilizado.

4.3.1 L1 propiedad división de enteros en desigualdad menor o igual que

\forall integer a, integer b, integer c; ((c>0) && (a>=0) && (a <= b*c)) ==> ((a/c) <= b)

$$\forall\, a, b, c \in \mathbb{Z}, c > 0, a \geq 0, a \leq b * c \rightarrow \frac{a}{c} \leq b$$

4.3.2 L2 propiedad división de enteros en desigualdad menor que

\forall integer a, integer b, integer c; ((c>0) && (a>=0) && (a < b*c)) ==> ((a/c) < b)

$$\forall\, a, b, c \in \mathbb{Z}, c > 0, a \geq 0, a < b * c \rightarrow \frac{a}{c} < b$$

4.3.3 L3 propiedad división de enteros en desigualdad menor o igual que

\forall integer a b c; (((c>0) && (a>=0) && (b>=0) && (a*c<=b)) ==> (a <= b/c))

$$\forall\, a, b, c \in \mathbb{Z}, c > 0, a \geq 0, a * c \leq b \rightarrow a \leq \frac{b}{c}$$

4.4 Métodos auxiliares

Los algoritmos de umbralización hacen uso de una serie de métodos auxiliares para su implementación. Para poder verificar los algoritmos es necesario haber verificado en primer lugar todos los métodos auxiliares utilizados.

4.4.1 countPixel

int countPixel(int h[])

La finalidad de este método es la de contar el número de píxeles de una imagen representada por el histograma *h*. El método va a ser de utilidad en varios algoritmos que utilizan el número de píxeles de una imagen, denotado como NM. Para calcular el número de píxeles lo que hay que hacer es sumar el número de píxeles que tienen cada intensidad.

Este método está implementado con un bucle que recorre todas las componentes del vector sumándolas para calcular el número total de píxeles.

Para verificar este método he hecho uso de la siguiente especificación.

Precondición

1. El vector debe ser no nulo
2. Todas sus componentes deben ser mayores o iguales que 0

Postcondición

1. El resultado debe ser igual a $\sum h[i]$
2. El resultado debe ser positivo

JML permite utilizar la palabra clave *sum* para especificar la postcondición y precondición. La especificación de la postcondición quedaría de la siguiente forma *\result == \sum int i; 0<=i<h.length;h[i].* Al tratar de ejecutar Krakatoa nos sale un error avisándonos de que la palabra clave *\sum* no es válida. Al comprobar la referencia de Krakatoa se indica que tanto *\sum*, *\product*, *\max* y *\min* no están soportados por Krakatoa. Su limitación se debe a que estas expresiones solo están

definidas cuando las variables están limitadas a un conjunto finito. Esta ha sido la razón que me ha llevado a haber definido el bloque axiomático *sum*.

Para hacer la verificación con Krakatoa necesitamos especificar un invariante y un variante. El invariante en nuestro caso cumple las siguientes condiciones:

1. $0 \leq j \leq h.length$
2. $s = sum(v, 0, j)$
3. $s \geq 0$

La que resulta más interesante es la condición 2 ya que es la que marca en cada paso cuánto vale la suma acumulada. Si se consigue verificar que la condición es invariable durante la ejecución el programa, entonces cuando $j = h.length$, la primera condición de la postcondición quedará probada. La segunda condición queda probada puesto que si siempre sumamos la componente i-ésima y en la precondición hemos exigido que todas las componentes son positivas el resultado de la suma será siempre positivo.

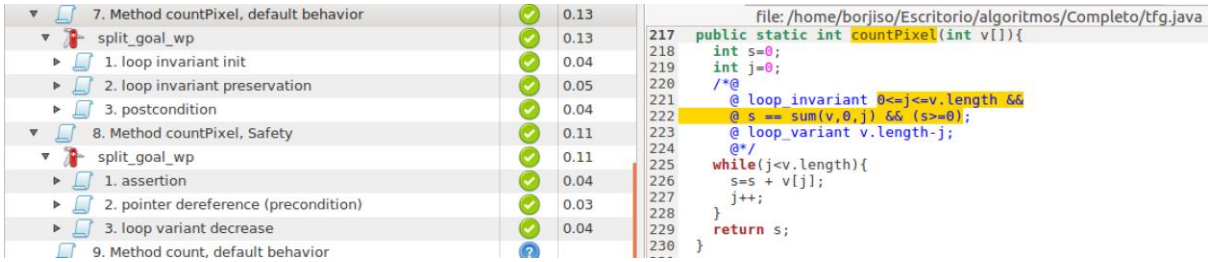


Ilustración 17: Algoritmo countPixel verificado.

4.4.2 count

int count(int[] h, int lo, int hi)

Este método es una generalización del método countPixel en el que se cuentan los píxeles dentro de un intervalo concreto de intensidades. Este método es utilizado por algoritmos que hacen particiones, como el *isodata*, y necesitan saber el número de píxeles en esas regiones.

La implementación es también similar. Se hace uso de un bucle, *for* en este caso, para recorrer las componentes entre *lo* y *hi*. Esta implementación incluye tanto a *lo* como *hi*. Lo primero que se hace es comprobar que los valores de *lo* y *hi* no son anómalos obligando a que *lo* sea mayor o igual que 0 y *hi* sea menor que la longitud del vector. Si no se cumplen estas condiciones lo que se hace es sustituir *lo* por 0 y la longitud del vector por -1 respectivamente.

En el caso de este método la especificación quedaría de la siguiente forma:

Precondición

1. Las componentes del vector deben ser mayores o iguales que 0
2. El vector debe ser no nulo
3. *h*i debe ser mayor que 0
4. *hi* debe ser menor que la longitud del vector
5. *lo* debe ser mayor que 0
6. La longitud del vector debe ser mayor que 0
7. La longitud del vector debe ser menor o igual que 256

Postcondición

1. El resultado debe ser igual a $\sum_{i=lo}^{hi} h[i]$
2. El resultado debe ser mayor o igual que 0

La condición 7 de la precondición no viene impuesta por la implementación original del método, sino que viene impuesta por necesidades de los algoritmos que utilizan este método.

Para poder hacer la especificación de este método he tenido que hacer una única modificación en el código. Dicha modificación ha sido la de declarar el índice *int i* antes del bucle. Esto se hace para que la variable sea visible para Krakatoa fuera del bucle puesto que va a formar parte del invariante.

A pesar de la similitud del código con el método *countPixel*, la especificación del invariante para hacer la demostración ha sido bastante más compleja. El invariante tiene las siguientes condiciones:

1. $lo \leq hi \rightarrow i \leq hi + 1$
2. $lo \leq i$
3. $cnt = sum(h, lo, i)$
4. $(lo > hi) \rightarrow (cnt = 0)$
5. $cnt \geq 0$

4.4.3 maxIntensityComponent

int maxIntensityComponent(int[] h)

Este método auxiliar calcula la intensidad máxima de una imagen a partir de su histograma. Es decir, lo que hace es buscar la mayor componente del vector tal que o sea la última componente o todas las componentes a su derecha son 0.

Tanto este método como el método *minIntensityComponent(int[] h)* han sido desarrollados para ser utilizados en el algoritmo *midRange* que forma parte de la familia de selección simple de umbral.

La implementación del algoritmo es propia puesto que el algoritmo *midRange* había sido planteado de forma teórica pero no ha sido implementado. El algoritmo realmente consiste en el típico método de búsqueda en un vector. Lo recorre hasta que encuentra una componente mayor que 0 y devuelve su índice. En caso de que todas las componentes sean 0, lo que quiere decir que la imagen que representa no tiene píxeles, entonces se devuelve 0. Para saber si se ha encontrado algún componente con valor distinto de 0 se hace uso de una variable auxiliar *máximo* que se inicializa a -1. El vector se recorre de derecha a izquierda buscando el primer componente distinto de 0.

La especificación propuesta para este algoritmo ha sido:

Precondición

1. El vector debe ser no nulo
2. Sus componentes deben ser mayores o iguales que 0
3. La longitud del vector debe ser mayor que 0

Postcondición

1. El resultado debe ser mayor que 0 y menor que la longitud del vector
2. Si el resultado es mayor que -1 entonces el resultado debe cumplir los axiomas de *isMaxIntensity*
3. Si todas las componentes del histograma son 0 entonces el resultado debe ser 0

El invariante en este caso está formado por las siguientes condiciones:

1. $0 \leq i < h.length$
2. $-1 \leq max < h.length$
3. $(\forall j \in \mathbb{Z}, ; 0 \leq j \leq h.length; h[j] = 0) \rightarrow max = -1$

Si el máximo es -1 al haber recorrido todo el vector, entonces se devuelve 0.

4.4.4 minIntensityComponent

int minIntensityComponent(int[] h)

Este método auxiliar calcula la intensidad mínima de una imagen a partir de su histograma. Es decir, lo que hace es buscar el menor componente del vector distinto de 0.

La implementación del algoritmo es propia por las mismas razones que en el algoritmo de intensidad máxima. Para saber si se ha encontrado algún componente con valor distinto de 0 se hace uso de una variable auxiliar *mínimo* que se inicializa a -1. El vector se recorre de izquierda a derecha buscando el primer componente distinto de 0.

La especificación propuesta para este algoritmo ha sido:

Precondición

4. El vector debe ser no nulo
5. Sus componentes deben ser mayores o iguales que 0
6. La longitud del vector debe ser mayor que 0

Postcondición

4. El resultado debe ser mayor que 0 y menor que la longitud del vector
5. Si el resultado es mayor que -1 entonces el resultado debe cumplir los axiomas de *isMinIntensity*
6. Si todas las componentes del histograma son 0 entonces el resultado debe ser 0

El invariante de este método es:

1. $0 \leq i < h.length$
2. $-1 \leq max < h.length$
3. $(\forall j \in \mathbb{Z}; 0 \leq j \leq h.length; h[j] = 0) \rightarrow min = -1$

Si el mínimo es -1 al haber recorrido todo el vector, entonces se devuelve 0.

4.4.5 doubleToInt

int doubleToInt (double x)

Este algoritmo ha sido utilizado para sustituir los *cast* propios de Java que utilizaban los algoritmos de umbralización. He hecho uso de la implementación de Ana que forma parte de los ejercicios realizados en la asignatura "*Especificación y Desarrollo de Sistemas de Software*". El algoritmo hace uso de una variable inicializada a cero a la que se le va sumando 1 mientras sea menor o igual que el número que queremos castear. La igualdad al estar comparando un entero con un doble correspondería a que $x - i$ sea menor que 1. Este método solo sirve para enteros positivos.

La precondición y postcondición del método son:

Precondición

1. El número que quiero castear debe ser mayor que cero y menor que 256*1

Postcondición

1. El resultado debe ser mayor que x-1 y menor o igual que x
2. el resultado debe ser igual a doubletoint(x)

El *doubletoint(x)* de la condición 2 de la postcondición es la función lógica definida anteriormente. La precondición se establece como 256*1. Esta condición se ha definido así porque no funcionaba en Krakatoa si se ponía directamente 256.

4.5 Algoritmos de umbralización

Una vez especificados y verificados los métodos auxiliares podemos pasar a especificar y verificar los algoritmos estudiados.

4.5.1 Selección simple de umbral

4.5.1.1 Media

double mean(int[] h, int lo, int hi)

El método de la media es utilizado como un método auxiliar dentro del algoritmo *isodata*. Esta es la razón de que la cabecera del método tenga los parámetros *lo* y *hi* que permiten calcular la media de un intervalo concreto dentro del vector que representa el histograma. Sin embargo, como también puede ser utilizado directamente como un umbral, en la especificación he tenido en cuenta las condiciones necesarias para que sea válido también para ser utilizado como umbral.

Puesto que estamos trabajando con un histograma, el algoritmo hace uso de dos variables auxiliares que representan la suma de píxeles total hasta la iteración *i* y la suma del número de píxeles por su intensidad. Así en nuestro algoritmo la variable *cnt* sería igual a $\sum_{n=lo}^{i} h[n]$ y *sum* sería $\sum_{n=lo}^{i} h[n] \cdot n$.

Puesto que la media estaría calculada por sum/cnt, el algoritmo antes de calcular y devolver la media comprueba si *cnt* es mayor que 0. Si ese es el caso se devuelve $(double)sum/cnt$. Se debe hacer un casteo de *sum* al tipo double puesto que como ambas variables son de tipo entero, Java realizaría una división de enteros truncando el resultado en caso de que hubiera decimales en el resultado. Si por el caso contrario *cnt* es 0, entonces se devuelve como resultado de la media 0.

La especificación utilizada para el algoritmo ha sido:

Precondición

1. Todas las componentes del vector deben ser mayores o iguales a 0
2. El vector debe ser no nulo
3. *lo* debe ser menor o igual que *hi*
4. *hi* debe ser menor que la longitud del vector
5. *hi* debe ser mayor que 0
6. La longitud del vector esté comprendida entre 0 y 256

Postcondición

1. Si todas las componentes en el intervalo *[lo, hi]* son 0, entonces el resultado es 0

2. Si alguna componente en el intervalo *[lo, hi]* es distinta de cero entonces el resultado es igual a sumHist(h,lo,hi+1)/sum(h,lo,hi+1) y el resultado estará comprendido entre *lo* y *hi*
3. El resultado estará comprendido entre 0 y *hi*

El invariante propuesto para este algoritmo es la siguiente:

1. $cnt \geq 0$
2. $sum \geq 0$
3. $lo \leq i \leq hi + 1$
4. $\forall i \in \mathbb{Z};\ (0 \leq i < h.length) ==> (i \cdot h[i] \geq 0)$
5. $cnt\ =\ sum(h, lo, i)$
6. $sum\ =\ sumHist(h, lo, i)$
7. $lo \cdot cnt \leq sum$
8. $sum \leq hi \cdot cnt$

Los demostradores por si solos no son capaces de demostrar que la media va a estar siempre entre *lo* y *hi*. Para demostrarlo ha sido necesario hacer uso del lema número 3 y el uso de asserts. Estos asserts verifican que:

1. i*h[i] <= hi*h[i]
2. sum <= sum + hi*h[i]
3. lo*h[i] <= i*h[i]

Y con estos asserts se puede verificar el invariante y posteriormente la postcondición. Todos estos asserts han sido necesarios para verificar que el resultado siempre va a estar entre el índice más bajo y el más alto. Esto no sería necesario para verificar la media, aunque sí que acotaría mejor el resultado. En este caso ha sido necesario añadirlo para poder utilizar este método en el algoritmo isodata. Si no lo añadíamos no se podía cumplir la precondición de otros métodos utilizados en dicho algoritmo.

4.5.1.2 Mediana

int median(int[] h)

La mediana representa el valor central del conjunto de datos. En nuestro caso, se trataría de la intensidad que deja a su izquierda la mitad de los píxeles de la imagen. La mediana, al igual que la media puede utilizarse también como un umbral.

/Users/borjiso/Desktop/code.java

```
3    @ requires vectorPositivo(h,h.length) && h!=null && (h.length > 0) && (sum(h,0,h.length-1)>0);
4    @ ensures (0 <= \result < h.length)
5    @ && ((sum(h,0,\result)) <= (sum(h,0,h.length-1)/2) <= (sum(h,0,\result+1)) );
6    @*/
7  public static int median(int[] h) {
8    int K = h.length;
9    int N = countPixel(h);
10   int m = N / 2;
11   int i = 0;
12   int sum = h[0];
13   /*@
14     @ loop_invariant (0 <= i <= K-1) && ((sum == sum(h,0,i+1))) && (0 <= sum - h[i] <= m);
15     @ loop_variant (h.length -i);
16     @*/
17   while (sum <= m && i < K) {
18     //@ assert m >= sum;
19     i++;
20     sum += h[i];
21   }
22   return i;
23 }
```

Ilustración 18: Código del algoritmo median

Este método también está implementado para aplicarse sobre un histograma. Esto quiere decir que el parámetro *h* representa el histograma de una imagen. Para calcular este valor se recorre el histograma sumando sus componentes mientras que la suma sea menor o igual que la mitad de los píxeles que tiene la imagen. El algoritmo también impone una condición de parada del bucle y es que dejaremos de sumar las componentes cuando ya no queden componentes que sumar. Si nos fijamos en la estructura, realmente esto solo podría ocurrir si tenemos la imagen vacía. Si eliminamos las asignaciones y ponemos los valores directamente en el bucle nos quedaría como condición del bucle $sum \leq \frac{countPixel(h)}{2} \wedge i < h.length$. Además, sabemos que para cada iteración se debe cumplir que $sum_{i-1} \leq sum_i$. El único caso entonces en el que se puede dar que *i* alcance la longitud del vector y *sum* sea menor o igual que *sum/2* es si la imagen no tiene píxeles.

Para especificar el algoritmo debemos tener en cuenta también que este método está haciendo uso del método auxiliar *countPixel.* Este método hace uso de *h* y observando el código se puede ver que no se hacen transformaciones sobre *h* por lo que la precondición de *countPixel* se cumpla, su precondición deberá aparecer tal cual en la precondición de la mediana. Así que teniendo en cuenta todo esto la especificación del algoritmo quedaría de la siguiente manera:

Precondición

1. Todas las componentes del vector deben ser mayores o iguales a 0
2. El vector debe ser no nulo
3. La longitud del vector debe ser mayor que 0
4. La suma de las componentes del vector debe ser mayor que 0

Postcondición

1. El resultado deberá estar entre 0 y la longitud del vector
2. $\sum_{i=0}^{mediana} h[i] \leq \frac{\sum_{i=0}^{h.length} h[i]}{2} \leq \sum_{i=0}^{mediana+1} h[i]$

Como podemos ver por la postcondición, este método se puede utilizar con cualquier vector represente o no una imagen. Esta especificación nos permite tener un algoritmo más generalizado útil

no solo para la umbralización de imágenes si no para cualquier otra situación que utilice un histograma.

Las condiciones 3 y 4 de la precondición son necesarias para poder asegurar que no se va a producir durante la ejecución un *"java.lang.ArrayIndexOutOfBoundsException".* Si no las ponemos Krakatoa nos avisará ya que no podrá validar la parte *safety* del algoritmo.

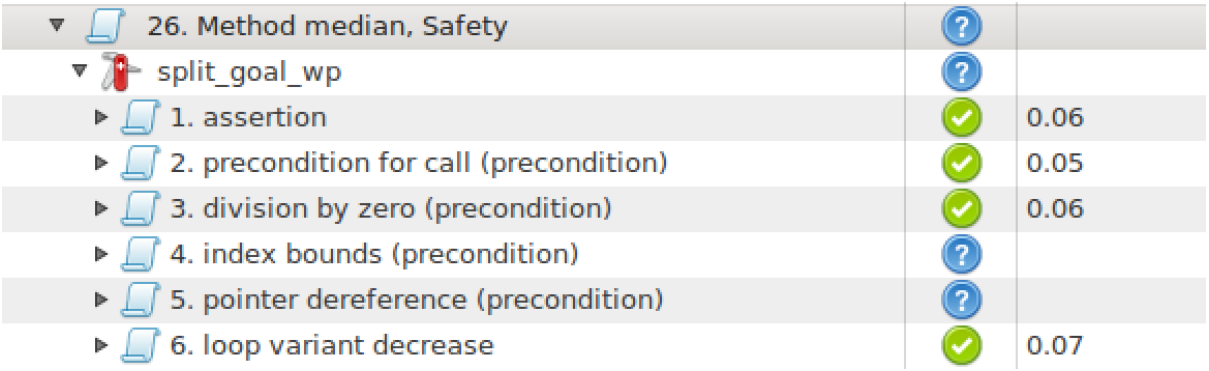


Ilustración 19: Condiciones safety del algoritmo median

Como podemos ver en la imagen, no se consigue verificar ni el *index bounds* ni el *pointer dereference* así que en tiempo de ejecución podrían aparecer la excepción ya mencionada y *NullPointerException*. La primera prueba comprueba que no se accedan a componentes de un vector fuera del rango válido. La segunda comprueba que no se accede a referencias nulas.

Este error podríamos provocarlo si tratamos de calcular la mediana de un vector con 0 elementos. Al tratar de ejecutar el fichero error.java obtenemos el error mencionado.

El invariante propuesto para el bucle es:

1. $0 \leq i \leq K - 1$
2. $sum = sum(h, 0, i + 1)$
3. $0 \leq sum - h[i] \leq m$

4.5.1.3 Umbral Percentil

int quantileThreshold (int[] h, int cu)

Cuando calculamos la mediana estamos buscando el valor que deja a su izquierda la mitad de las observaciones, es decir, el valor que divide las observaciones en dos partes. Los percentiles son los valores que permiten dividir las observaciones en partes iguales. Así la mediana sería un caso particular en el que se estaría calculando el percentil 50.

Originalmente la implementación hacía uso de una variable que especificaba el percentil. Para facilitar la corrección modifiqué esta variable y le añadí un parámetro a la función que haría el mismo trabajo que la variable.

Además del percentil que se quiere calcular, se hace uso también de un vector que representa el histograma de la imagen.

Puesto que se trata de un algoritmo que es la generalización de la mediana, la implementación debería haber sido prácticamente idéntica. Pero no es el caso. El algoritmo hace uso de un bucle *while* para recorrer el vector hasta que la suma acumulada se corresponda con el percentil deseado o hasta que sé que no queden más componentes. Finalmente se devuelve la intensidad que se corresponde con

el percentil o -1 si se ha producido algún error y la suma acumulada es mayor que el número de píxeles de la imagen. Esta última es la diferencia entre ambos.

La diferencia entre ambos viene motivada porque la mediana realmente es utilizada como método auxiliar y no directamente como umbral. Entre los algoritmos implementados en el libro sí que aparece un *MedianThresholder.* Esta clase lo que hace es heredar de la clase *QuantileThresholder*, que es la que implementaba el algoritmo del percentil, y llamar al constructor super() con parámetro 0.5. Es decir, simplemente estaba encapsulando un caso concreto en una clase propia.

```
1  /*@
2    @ requires VectorPositivo(h,h.length) && h!=null && (h.length > 0) && (sum(h,0,h.length-1)>0) && (0<cu<1);
3    @ ensures (0 <= \result < h.length)
4    @ && ((sum(h,0,\result-1)) <= (sum(h,0,h.length-1) * cu) <= (sum(h,0,\result)));
5    @*/
6  public static int quantileThreshold (int[] h, int cu){
7    int K = h.length;
8    int N = countPixel(h);
9    double m = N * cu;
10   int i = 0;
11   int sum = h[0];
12   /*@
13     @ loop_invariant (0 <= i <= K-1) && ((sum == sum(h,0,i+1))) && (0 <= sum - h[i] <= m);
14     @ loop_variant (h.length -i);
15     @*/
16   while (sum < m && i < K) {
17     //@ assert m >= sum;
18     i++;
19     sum += h[i];
20
21   }
22   int q = (sum < N) ? i : -1;
23   return q;
24 }
```

Ilustración 20: Código algoritmo quantileThreshold

En primer lugar, se inicializan las variables auxiliares que almacenan el número de píxeles, la cantidad de píxeles que cumplen el percentil y se inicializan las variables de índice y la suma acumulada.

Como este método sí que estaba implementado directamente para usarse como umbral, se ha añadido una condición que permite que el método devuelva -1 en caso de que ocurra algún error. Si nos fijamos en esta condición, el error esperado es que la suma acumulada que hemos calculado supere al número total de píxeles que tiene la imagen. Este caso es un poco absurdo y la única manera de que tenga lugar es si el método auxiliar utilizado que calcula el número de píxeles de la imagen está mal implementado, o que el bucle que va realizado la suma de las componentes esté mal. En cualquiera de los dos casos se puede ver una desconfianza en el propio código. Podemos decir que, si la implementación del código se hubiese realizado a partir de una correcta especificación del algoritmo, como por ejemplo el uso de precondición y postcondición, y su posterior verificación, no se tendría por qué haber usado esta guarda.

La especificación propuesta es la siguiente:

Precondición

1. El vector debe ser no nulo
2. Todas las componentes del vector deben ser mayores o iguales que 0
3. La longitud del vector debe ser mayor que 0
4. Al menos una componente del vector debe ser mayor que 0
5. El percentil esté entre 0 y 1

Postcondición

1. El umbral devuelto debe ser un valor comprendido entre 0 y la longitud del vector
2. $\sum_{i=0}^{percentil} h[i] \leq \sum_{i=0}^{h.length} h[i] \cdot cu \leq \sum_{i=0}^{percentil+1} h[i]$

Con la precondición 4 evitamos que el método pueda devolver -1 ya que solo se puede dar la condición $sum \geq n$ cuando todas las componentes del vector sean 0. He puesto esta precondición porque es la única forma de que ocurra eso, aparte de los posibles errores de implementación en el propio método o métodos auxiliares como antes he comentado, es que la imagen no tenga píxeles y eso es muy poco probable que ocurra en el mundo real.

4.5.1.4 MidRange

int midRange (int[] h)

La implementación del método es muy sencilla puesto que lo que se hace es calcular la media entre los dos valores devueltos por los métodos auxiliares m*inIntensityComponent* y *maxIntensityComponent* llamados con el parámetro *h.* Como en los casos anteriores el parámetro *h* representa el histograma de la imagen.

Para establecer la precondición del algoritmo debemos utilizar las precondiciones de los métodos auxiliares. Así la especificación del algoritmo es:

Precondición

1. El vector debe ser no nulo
2. Todas las componentes deben ser mayores o iguales a 0
3. La longitud del vector debe ser mayor que 0

Postcondición

1. El resultado debe estar entre 0 y la longitud del vector
2. $\exists\ integer\ a, integer\ b, (0 \leq a < h.length)\ \wedge\ (0 \leq b < h.length)\ \wedge\ (isMaxIntensity(h, a)\ \wedge\ isMinIntensity(h, b)\ \wedge\ (\backslash result = (a + b)/2))$

La condición 2 de la postcondición es tan compleja porque hemos definido *isMaxIntensity* y *isMinIntensity* como predicados y no como bloques axiomáticos. Podría haber hecho un predicado que me permitiera simplificarlo, pero he decidido especificarlo directamente puesto que esa expresión no va a utilizarse más veces.

Dado que la implementación es tan sencilla y no hay bucles no han sido necesarios ni invariantes ni variantes. Además, como *isMaxIntensity* y *isMinIntensity* forman parte de la postcondición de los métodos auxiliares la verificación de que $\backslash result == (a + b)/2$ es inmediata.

4.5.2 Isodata

int isodataThreshold(int[] h, int MAX_ITERATIONS)

Como hemos visto en el estudio de este algoritmo se trata de un algoritmo iterativo. En los algoritmos anteriores hemos podido especificar las características que debe cumplir el umbral calculado. Pero en esta ocasión es muy difícil determinar qué condiciones debe cumplir. Lo poco que podemos decir con seguridad es que el umbral debe estar comprendido entre 0 y 255.

Como hemos visto antes, el algoritmo calcula el umbral haciendo la media de los centros de dos subconjuntos creados a partir de un valor inicial. Así que podríamos incluso decir que el valor del

umbral va a estar comprendido entre esos dos centros. Aun así, nos seguiría quedando una postcondición muy débil.

Con la aproximación anterior nos damos cuenta de lo que realmente nos puede ser útil es especificar cuál debe ser el valor del umbral en cada iteración y como postcondición establecer que el resultado deber ser igual a hacer el cálculo del umbral n veces. Para facilitar esta especificación hemos hecho con anterioridad el bloque axiomático que describe cuánto vale el umbral en cada iteración. Además, hemos visto que también tiene en cuenta los casos raros del algoritmo.

La especificación propuesta es la siguiente:

Precondición

1. El vector debe ser no nulo
2. Todas las componentes del vector deben ser mayores o iguales que 0
3. La longitud del vector debe ser 256
4. El límite de iteraciones debe ser mayor que 0

Postcondición

1. El resultado debe ser el obtenido por el bloque axiomático.
2. El resultado debe estar comprendido entre 0 y 255.

He tenido que modificar el código del algoritmo para adaptarlo a la forma en la que estamos haciendo la verificación.

La primera modificación realizada ha consistido en pasar como parámetro el límite de iteraciones del algoritmo. Haciendo esto conseguimos una especificación más general y evitamos el riesgo de que solo sea válida la verificación para el límite escogido.

La siguiente modificación consiste en sustituir el *cast* propio de Java por el uso del método auxiliar que hemos creado y verificado.

He declarado las variables auxiliares fuera del bucle *do while* para poder acceder a ellas desde el invariante. Estas variables han sido *int nB, nF* y *double meanB, meanF.* Todas las variables han sido inicializadas a 0. Esta inicialización no entra en conflicto con el código original ya que el valor devuelto por los métodos utilizados para inicializar las variables dentro del bucle y que han sido previamente verificados son mayores o iguales que 0.

El cambio más importante, puesto que afecta al propio algoritmo, es que he eliminado una de las condiciones de permanencia en el bucle. En concreto la que rompía la ejecución del bucle si el umbral calculado se estabilizaba. Realmente esta modificación lo único que va a conseguir es que se hagan iteraciones inútiles puesto que el nuevo umbral calculado será igual al anterior y la ejecución se mantendrá hasta alcanzar el límite de iteraciones. Este cambio lo he realizado porque simplifica la verificación del algoritmo y dada la falta de tiempo, lo más probable es que no pudiese terminarla a tiempo para la entrega.

Con el código modificado y la especificación de la precondición y postcondición realizada podemos pasar a la verificación.

Para la verificación he hecho uso de una variable fantasma *a* que guarda el valor de la media de la imagen puesto que la variable *q* en la que se almacena por primera vez, es actualizada en cada iteración y se pierde. Este dato es necesario puesto que para poder verificar la postcondición es

necesario verificar que se cumple para *i* iteraciones el bloque axiomático y dicho bloque necesita en primer lugar la media como umbral inicial.

En este caso para probar el algoritmo he tenido que hacer uso de algunos. Es el caso en el que se comprueba si al llamar al bloque isodata con un límite de iteraciones 0, este devuelve el umbral inicial.

Llegado a este punto el invariante del método ha quedado así:

1. $1 \leq i \leq MAX\_ITERATIONS$
2. $0 \leq q < K$
3. $0 \leq q\_ < K$
4. $nB \geq 0$
5. $nF \geq 0$
6. $meanB \geq 0$
7. $meanF \geq 0$
8. $(((meanB + meanF)/2) - 1 < q \leq (meanB + meanF)/2) \,||\, (q = sumHist(h, 0, h.length)/sum(h, 0, h.length))$
9. $(i > 0) \rightarrow ((nB = sum(h, 0, q\_ + 1)))$
10. $(i > 0) \rightarrow ((nF = sum(h, q\_ + 1, h.length)))$
11. $(meanB = sumHist(h, 0, q\_ + 1)/sum(h, 0, q\_ + 1))$
12. $(meanF = (sumHist(h, q\_ + 1, h.length))/(sum(h, q\_ + 1, h.length)))$
13. $(doubletoint((((sumHist(h, 0, q\_ + 1)/sum(h, 0, q\_ + 1)) + ((sumHist(h, q\_ + 1, h.length))/(sum(h, q\_ + 1, h.length))))/2.0)) = doubletoint((meanB + meanF)/2.0))$
14. $q = isodata(h, h.length, q\_,1)$
15. $q\_ = isodata(h, h.length, a, i - 1)$
16. $q = isodata(h, h.length, a, i)$

Este invariante es el más largo de todos los utilizados y se debe en parte a la cantidad de variables que intervienen durante la ejecución del algoritmo, pero también en la complejidad del bloque axiomático utilizado.

Dentro del bucle hay un condicional que comprueba $nB = 0 | nF = 0$. Si se da esa condición se devuelve -1. Cuando sucede esto, para el resto del bucle no debería darse el caso en el que *sum(h,0,q_+1)* o *sum(h,q_+1,h.length)* sean 0. Pero esto no sucede así y es necesario especificar en varios de los *asserts* que solo se cumplen si $nB > 0 \wedge nF > 0$. Estos asserts han sido necesarios para verificar las condiciones del invariante.

La estrategia seguida para la verificación de la postcondición es una especie de inducción. Si podemos probar $q == isodata(h, h.length, q\_,1)$, $q\_ == isodata(h, h.length, a, i - 1)$ entonces deberíamos conseguir probar $q == isodata(h, h.length, a, i)$ y con ello se probaría la postcondición. Los demostradores automáticos no han sido capaces de verificar este último paso por lo que la verificación del algoritmo no ha sido satisfactoria. Para tratar de verificarlo hemos optado por verificar este paso con demostradores interactivos. En nuestro caso hemos utilizado Isabelle.

Puesto que esta es la única condición que queda sin probar, si se probase con Isabelle la corrección del algoritmo quedaría probado. Krakatoa nos permite hacer uso de terceras herramientas para demostrar las pruebas que se generan. Para ello debemos haberlo instalado y configurado siguiendo las instrucciones del sitio [14]. Tras abrirlo con Isabelle, Jose ha sido capaz de demostrar la validez y por tanto el algoritmo ha sido verificado.

4.6 Binarización de una imagen

A demás de los algoritmos de umbralización, he implementado los siguientes algoritmos relacionados con la binarización de imágenes. Estos métodos han sido:

1. Frecuencia de aparición de un elemento en un vector
2. Generación de un histograma a partir de una imagen
3. Binarización de una imagen dado un umbral

Los métodos han sido implementados desde 0.

4.6.1 Frecuencia

frequency(int v[], int x)

Este método calcula la frecuencia de aparición del elemento *x* en el vector *v*. Es utilizado como método auxiliar del método que genera el histograma de una imagen representada por un vector de enteros.

El método está implementado con una variable que cuenta las apariciones del elemento buscado y un bucle que recorre el vector completo. Si $v[i] = x$ entonces se aumenta el valor de la variable *appearance*.

La especificación del algoritmo ha sido:

Precondición

1. El vector debe ser no nulo
2. La longitud del vector debe ser mayor que 0
3. El elemento a contar debe ser mayor o igual que 0

Postcondición

1. El resultado debe ser igual al obtenido por el bloque axiomático *count*

En cuanto al invariante, el mismo ha sido:

1. $0 \leq i \leq v.length$
2. $appearance = count(v, 0, i, x)$

4.6.2 Histograma

void histogram(int I[], int h[])

Este método calcula el histograma *h* de una imagen *I*. La implementación se ha hecho recorriendo las intensidades del histograma con un bucle *for* y poblando el histograma con las frecuencias encontradas para cada intensidad en la imagen.

La especificación a partir de la cual se ha implementado el método ha sido la siguiente:

Precondición

1. La imagen debe ser no nula
2. La longitud de la imagen debe ser mayor que 0
3. La imagen debe ser una imagen en escala de grises
4. El histograma debe ser no nulo
5. La longitud del histograma debe ser 256

6. Todas las componentes del histograma deben ser 0

Postcondición

1. La longitud del histograma no debe cambiar
2. Para toda intensidad entre 0 y 255, la componente de la intensidad *i* debe ser igual a la frecuencia de aparición *i* en la imagen *I*

Si nos fijamos en la precondición, la condición 6 podría ser obviada si se hubiera pensado en el algoritmo devolviendo un nuevo histograma creado dentro del método. Aunque se hubiera realizado de esta manera y sabiendo que java inicializa las componentes de un vector de enteros a 0, no hubiera sido posible ya que para Krakatoa un nuevo vector de enteros no tiene inicializadas las componentes a 0 y las considera como nulas.

El invariante utilizado para la verificación ha sido:

1. $0 \leq i \leq 256$
2. $b = a$
3. $\forall j \in \mathbb{Z};\ (0 \leq j < i) \rightarrow\ (h[j] = count(I, 0, I.length, j))$

La variable *b* de la condición 2 es una variable fantasma utilizada para comprobar que durante la ejecución del bucle el valor de *a* no cambiaba. He tenido que hacer esto porque durante el desarrollo del método han ocurrido cosas raras. En primer lugar, la condición 2 de la postcondición no se conseguía verificar ni siquiera con el invariante. Así que hice uso de *asserts* para comprobar que se estaba haciendo bien el conteo de la frecuencia de los píxeles. Al hacer esta comprobación pude observar un comportamiento extraño. Como se puede ver en el código hice uso de una variable *a* que contenía la frecuencia para cada intensidad *i*. Posteriormente se hace una asignación de *a* en *h[i].* Los *asserts* hasta este punto conseguían verificarse, pero cuando se llegaba al punto de la asignación no. Es decir, al asignar el valor a la componente correspondiente del vector, dicha componente no cumplía las condiciones que cumplía la variable *a.* La solución ha sido utilizar *SeparationPolicy = Regions.* Esto permite hacer un análisis estático de las referencias. Lo que supone que cuando se muta una referencia, las propiedades que cumple *a* al ser su valor asignado a *h[i]* se van a seguir cumpliendo. Si no dispusiese de esta instrucción, tendría que haber creado un lema que definiera esta propiedad.

4.6.3 Binarización

void binarize(int I[], int q)

Este método transforma una imagen en escala de grises *I* en una imagen en blanco y negro usando como umbral el valor *q.* Para hacerlo se recorre la imagen con un bucle y se comprueba si la intensidad del píxel es mayor o menor o igual que el umbral dado. Si la intensidad es menor o igual que el umbral el píxel es establecido a 0 y si se da el caso contrario entonces se establece a 1.

La especificación utilizada para la implementación del método ha sido:

Precondición

1. La imagen debe ser no nula
2. La longitud de la imagen debe ser mayor que 0
3. La imagen debe ser en escala de grises
4. El valor del umbral debe estar comprendido entre 0 y 255

Postcondición

1. Todos los píxeles de la imagen deben ser 0 o 1
2. Para todo píxel de la imagen si el valor inicial era menor o igual que el umbral ahora debe ser un 0 y si era mayor ahora debe ser un 1

El invariante del bucle ha sido:

5. $0 \leq i \leq I.length$
6. $\forall k \in \mathbb{Z};\ (i \leq k < I.length) \rightarrow (\backslash at(I[k], Pre) = I[k])$
7. $\forall j \in \mathbb{Z};\ (0 \leq j < i \rightarrow at(I[j], Pre) \leq q \rightarrow I[j] = 0)))\&\&((\backslash at(I[j], Pre) > q) \rightarrow (I[j] = 1)))$

4.7 Problemas encontrados

Durante el desarrollo del trabajo han aparecido varios problemas de distinta índole. Estos problemas an sido tanto prácticos como teóricos.

Durante la verificación del algoritmo isodata se hice uso de un lema que era falso. Esto llevó a que los demostradores, haciendo uso de este, validaran las condiciones y se consiguiera verificar el método. Cuando conseguí, aparentemente, verificar el algoritmo, en vez de enviárselo a los tutores para que pudieran comprobarlo, seguí trabajando en otras tareas. Posteriormente, cuando Jose revisó el código, se percató de que el lema era falso. Al eliminarlo, los demostradores ya no eran capaces de verificar el algoritmo y tuve que continuar con él. Esta ha sido una de las causas del retraso de la entrega.

Uno de los problemas han sido las limitaciones del subconjunto JML utilizado por Krakatoa. Debido a la falta de algunas utilidades como *\sum,* he tenido que dedicar tiempo al desarrollo de funciones auxiliares que suplieran estas limitaciones. Esto no ha supuesto finalmente mucho problema una vez he comprendido cómo se usaban los predicados y bloques axiomáticos. Además, ha supuesto la creación de una serie de algoritmos básicos que pueden ser aprovechados en un futuro. Estas funciones, al haber sido generalizadas, podrán utilizarse en más ámbitos y no solo en el procesamiento digital de imágenes como puede ser la verificación de funciones estadísticas.

Otro de los problemas que ha surgido se ha debido al utilizar Krakatoa de forma independiente a un entorno de desarrollo como Eclipse. Esto ha llevado a que los procesos de depuración de errores en las anotaciones no hayan sido sencillos. El error que nos devuelve el programa al tratar de ejecutar la herramienta sobre un código con errores tiene el siguiente aspecto:

```
borjiso@borja-TFG:~/Escritorio/algoritmos/Completo$ krakatoa tfg.java
Warning: tfg is not a subpackage of .
Warning: tfg_mean_saved is not a subpackage of .
Warning: tfg2 is not a subpackage of .
File "tfg.java", line 256, characters 46-47: syntax error (parse error in annota
tion)
```

Ilustración 21: parse error in annotation

En este caso se muestra el error producido al haber un paréntesis de más en el código. Este ejemplo está bastante acotado, pero muchas veces pueden salir errores en los que el error está acotado entre 20 caracteres o más.

Además, este problema se ha visto agravado por dos situaciones. La primera ha sido el editor de código utilizado para el desarrollo del código. Este editor ha sido SublimeText3. SublimeText3 inserta tanto

el paréntesis de apertura como el de cierre cada vez que pulsas la tecla de paréntesis. Por ello tiene implementado un sistema por el cual, si cierras manualmente el paréntesis, en vez de duplicarse se corre el cursor sin insertar el nuevo paréntesis. Esto puede resultar muy útil cuando estás escribiendo el código. Lo que ocurre es que cuando estás realizando correcciones y hay muchos paréntesis, al añadir uno nuevo, porque te has dejado un par de paréntesis o quieres dar prioridad a una operación concreta, si ya hay un paréntesis a la derecha del cursor no se te va a insertar. Esto lleva a que hay más paréntesis de apertura que de cierre y se produce un error de *parseo*.

Por desgracia el error se da demasiado a menudo puesto que frecuentemente los predicados son demasiado complicados que requieren de muchos paréntesis y varias líneas de código.

```
1  //@assert ((nF>0)&&(nB>0))==>(q == (integer)(((sumHist(h,0,q_+1)/sum(h,0,q_+1) + (sumHist(h,q_+1,h.length))/(sum(h,q_+1,h.length))))/2));
```

Ilustración 22: Condición compleja de un bloque axiomático

Durante el desarrollo del código he tratado de utilizar un bloque axiomático para calcular la media de un vector. Sin embargo, al tratar de utilizarlo han aparecido errores relacionados con Jessie. Estos errores se debían al uso de *cast* dentro de los bloques axiomáticos.

Un problema bastante raro ha sido el nombre de uno de los bloques axiomáticos. Este bloque ha sido *sumHist.*

```
1  /*@ axiomatic Mean {
2    @   logic double mean{L}(int [] h, integer a, integer b);
3    @ axiom mean1{L} :
4    @ \forall int h [], integer a, integer b; ((sum(h,a,b+1)==0) && (b<h.length)) ==> (mean(h,a,b) == 0);
5    @ axiom mean2{L} :
6    @ \forall int h [], integer a, integer b; ((sum(h,a,b+1) > 0) && (b<h.length)) ==> (mean(h,a,b) == (sumHist(h,a,b+1)/(sum(h,a,b+1))));
7    @ axiom mean3{L} :
8    @ \forall int h [], integer a, integer b; (a>b) ==> (mean(h,a,b) == 0 );
9    @ axiom mean4{L} :
10   @ \forall int h[], integer a, integer b; (b>=h.length) ==> (mean(h,a,b)==mean(h,a,h.length-1));
11   @ axiom mean5{L} :
12   @ \forall int h[], integer a, integer b; (a<0) ==> (mean(h,a,b)==mean(h,0,b));
13   @ }
14   @*/
```

Ilustración 23: código de la función lógica mean

En la primera versión, el predicado *sumHist,* se llamaba *sumHis.* Cuando trataba de ejecutar Krakatoa obtenía el siguiente error.

```
borjiso@borja-TFG:~/Escritorio/algoritmos/Completo$ krakatoa tfg.java
Warning: tfg is not a subpackage of .
Parsing OK.
Typing OK.
Anomaly: cannot find logic symbol `sumHis'
[isMaxIntensity;\real_abs;sum;\int_max;\int_min;esMin;esComMin;isodata;isMinInte
nsity;esMax;\real_max;\cos;VectorPositivo;minInRange;\real_min;esComMax;]
File "java/java_interp.ml", line 363, characters 1-1:
Fatal error: exception File "java/java_interp.ml", line 363, characters 1-7: Ass
ertion failed
```

Ilustración 24: Error producido por el nombre sumHis

Aparentemente no se encontraba el predicado *sumHis.* Las causas más lógicas era pensar que estaba mal escrito o que no se podía usar un predicado dentro de otro. Tras descartar lo primero, y percatarme de que ya había hecho uso de predicados dentro de otros predicados empezamos a hacer pruebas. Al investigar el fichero *java/java_interp.ml,* en la línea que nos dice el error, solo se puede observar el código que imprime por pantalla ese error en caso de que se produzca un *Assertion failed*. Tras realizar diversos cambios, Ana descubrió que si se ponía otro nombre como *sum2* o *sumHis* no se producía el problema. Tras comprobar esto comprobamos si existía alguna limitación en el número de

caracteres de los nombres. Esto también lo descartamos. No hemos llegado a descubrir a qué se debe el error.

Hemos visto que Krakatoa genera unas condiciones de verificación a partir de las anotaciones JML. Estas condiciones pueden resultar de ayuda para tratar de averiguar qué condiciones podemos añadir al invariante o a los asserts para tratar de demostrar las pruebas. Sin embargo, la extensión de dichas condiciones y su complejidad no nos aportan nada si no contamos con experiencia en el uso de demostradores.

```
axiom H :
  left_valid_struct_intM v_21 0 usObject_v_21_21_alloc_table /\
   usNon_null_intM v_21 usObject_v_21_21_alloc_table /\
    usVectorPositivo v_21
     (offset_max usObject_v_21_21_alloc_table v_21 + 1) intM_intP_v_21_21
```

Ilustración 25: Condiciones generadas por Krakatoa

El fin de la especificación formal es el de dar una especificación del problema que pueda ser válida para cualquier implementación que se haga de un algoritmo. Esta especificación no siempre es fácil de definir como hemos podido ver en el trabajo desarrollado. En el ámbito del procesamiento digital de imágenes, y más en concreto el de la umbralización, es muy difícil especificar las condiciones que debe cumplir un buen umbral. Esto se debe a que la adecuación de ese umbral es a la imagen y no se puede generalizar para todas las imágenes. Como hemos visto, lo único que podríamos definir como un umbral correcto que nos sirva para cualquier imagen es que el valor de ese umbral va a estar siempre entre el valor de la intensidad mínima y el de la intensidad máxima. Esto nos obliga a centrar la especificación en una solución concreta y no en la especificación del problema.

Otro de los problemas que he tenido es el de trabajar con código ya implementado. Lo lógico sería implementar el código a partir de su especificación. En este caso he tenido que dar una especificación que se adecue a la implementación de los algoritmos. Esto ha supuesto que las especificaciones de los algoritmos se hayan visto modificadas numerosas veces, y dado que un cambio en la especificación del método puede suponer conflictos con la de los métodos que lo están utilizando, un pequeño cambio produce que se tengan que cambiar muchos otros métodos. Para evitar esto lo que se debe hacer es modificar las precondiciones de los métodos para hacerlas lo menos estrictas posibles y establecer las postcondiciones con implicaciones que consideren cómo debe ser el resultado para ciertas condiciones. Por ejemplo, en la media permitir que *lo* sea mayor que *hi* y en la postcondición establecer que resultado se obtendría si eso se cumple.

5. Conclusiones

La especificación formal es una herramienta muy potente que nos permite modelar problemas y crear soluciones correctas. Lo ideal sería crear esas soluciones a partir de la especificación, pero esto no siempre puede darse. Al usar la verificación formal para comprobar la corrección de un algoritmo podemos vernos en dos casos. Verificar algoritmos implementados por terceras personas y cuya especificación formal desconocemos, o verificar algoritmos que hemos implementado a partir de una especificación.

El primer caso es el que hemos tratado en este trabajo. Vistos los problemas encontrados, podemos ver que la especificación de algoritmos ya implementados puede verse influenciada por el código del que disponemos. Esto hace, de manera casi irremediable, que acabemos especificando qué es lo que hace el algoritmo y no las condiciones que debería cumplir el resultado del algoritmo.

También hemos podido ver como hay ocasiones en las que realizar la especificación de un problema resulta tan complicado que nos lleva a hacer condiciones demasiado generales. Debido a estas condiciones tan generales, no resulta de utilidad dicha especificación y acabamos por especificar el comportamiento del algoritmo.

En cuanto a las herramientas existentes, nos hemos centrado en el uso de demostradores automáticos. Hemos visto como hay muchas pruebas y condiciones que, siendo triviales para nosotros, para los demostradores resulta imposible de probar. También suelen tener problemas para probar enunciados cuya demostración requiere del uso de inducción. En este caso entra en juego el uso de demostradores interactivos como Isabelle que nos permiten guiar a los demostradores para realizar esta verificación.

6. Trabajo pendiente

En este trabajo he desarrollado una investigación que se ha quedado corta en muchos aspectos debido a la limitación temporal con la que contaba. Es por eso por lo que hay algunos temas tratados que se podrían profundizar en un futuro.

En este trabajo me he centrado en algoritmos de umbralización global dejando de lado los algoritmos adaptativos. Este sería una tarea a desarrollar partiendo de los conocimientos adquiridos en este trabajo.

Una mejora que se puede hacer sobre el trabajo es la de tratar de conseguir una precondición mínima para las especificaciones propuestas.

Puesto que estaba planteado inicialmente, aprender cómo especificar, desarrollar y verificar programas escritos con lenguajes funcionales y ver las ventajas que puedan tener frente a Java.

La especificación, desarrollo y verificación de los lenguajes escritos con lenguajes funcionales era una tarea inicial que por falta de tiempo no ha podido desarrollarse. En un futuro se podría realizar y estudiar sus posibles ventajas frente a la implementación con Java y su verificación.

7. Lecciones aprendidas

Uso de un diario

En un trabajo de investigación como este el uso de un diario puede resultar de gran utilidad. En el podemos apuntar las tareas que vamos realizando cada día. Cuando recurrimos a libros, artículos y otros recursos podemos apuntar las conclusiones que sacamos de ellos y la información necesaria para poder referenciar correctamente nuestro trabajo.

En mi caso, también he usado el diario para apuntar las pruebas que iba realizando y apuntar las razones de las modificaciones que llevaba a cabo.

También es muy útil para saber el punto en el que nos quedamos en la jornada anterior de trabajo.

Evitar los periodos largos de inactividad

Cuando dejamos durante un tiempo prolongado el desarrollo de un trabajo puede ocurrirnos que después nos resulte más complicado retomar el trabajo. Si nos ocurre esto por causas mayores, es mejor dedicar un poco de tiempo cada día a seguir trabajando que no dejarlo y retomarlo cuando las circunstancias lo permitan.

Máquinas virtuales

Para la realización de pruebas es recomendable hacer uso de máquinas virtuales. Esto nos va a permitir hacer uso de las utilidades que nos ofrecen, como el caso de las instantáneas. Las instantáneas nos permiten volver a un estado consistente en cualquier momento. En caso de que realizando pruebas o al hacer instalaciones y desinstalaciones produzcamos un fallo en la máquina, podremos recuperar la instantánea y seguir trabajando sin perder tiempo en arreglar nuestro equipo.

Otra gran ventaja es que nos permite recrear de forma sencilla las condiciones en las que hemos trabajado y así reproducir las pruebas que hagamos y los fallos que puedan surgir.

Uso de la nube

Utilizar un sistema de almacenamiento en la nube es esencial para salvaguardar nuestro trabajo. También nos permite el acceso a nuestro trabajo sin disponer de nuestro equipo habitual.

Si unimos el uso de la nube y las máquinas virtuales, conseguimos tener un acceso a nuestras herramientas de trabajo dese cualquier momento y evitar dificultades por averías o robos.

Paperline

Paperline [15] es una herramienta que permite gestionar las referencias a las fuentes de información que utilizamos para nuestras investigaciones. Permite almacenar todas las fuentes de información y generar referencias en diferentes estilos.

8. Bibliografía

[1] "ImageJ." n.d. ImageJ. Consultada 17 de julio de 2018. https://imagej.net/ImageJ.

[2] La Serna Palomino, Nora y Román Concha, Norberto Ulises. 2009. "Técnicas de Segmentación En Procesamiento Digital de Imágenes." Revista de Investigación de Sistemas de Informática http://revistasinvestigacion.unmsm.edu.pe/index.php/sistem/article/view/3299.

[3] "The Java Modeling Language (JML) Home Page." n.d. Consultada 24 de abril de 2018. http://www.eecs.ucf.edu/~leavens/JML//index.shtml.

[4] "Why: A Software Verification Platform." n.d. Consultada 27 de febrero de 2018. http://why.lri.fr/.

[5] "Krakatoa and Jessie: Verification Tools for Java and C Programs." n.d. Consultada 27 de febrero de 2018. http://krakatoa.lri.fr/.

[6] "Project – The KeY Project." n.d. Consultada 2 de marzo de 2018. https://www.key-project.org/about/project/.

[7] "Download – The KeY Project." n.d. Consultada 2 de marzo de 2018. https://www.key-project.org/download/.

[8] Ahrendt, Wolfgang, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. 2016. *Deductive Software Verification – The KeY Book: From Theory to Practice*. Springer.

[9] "Home | OpenJML." n.d. Consultada 12 de marzo de 2018. http://www.openjml.org/. "Installing OpenJML." http://www.openjml.org/documentation/installation.shtml.

[10] "OpenJML (ESC) @ rise4fun from OpenJML." n.d. Consultada 25 de julio de 2018. https://www.rise4fun.com/OpenJMLESC/.

[11] Manual, Reference. n.d. "The Krakatoa Verification Tool for JAVA Programs." http://krakatoa.lri.fr/krakatoa.pdf.

[12] Burger, Wilhelm, and Mark J. Burge. 2016. *Digital Image Processing: An Algorithmic Introduction Using Java*. Springer.

[13] "Krakatoa." n.d. Consultada 24 de abril de 2018. http://toccata.lri.fr/gallery/krakatoa.en.html.

[14] "Compilation, Installation." n.d. Consultada February 27, 2018. http://why3.lri.fr/doc/install.html.

[15] Chinn, Peggy L. 2016. "Paperpile and Google Docs." *Nurse Author & Editor* 26 (4): 4.